Students often tackle programming problems with a flawed understanding of what the problem is asking. Some pedagogies attempt to address this by encouraging students to develop examples in the form of input–output assertions (henceforth “functional examples”), independent of (and typically prior to) developing and testing their implementations. However, without an implementation to run examples against, examples are impotent and do not provide feedback. Consequently, students may be inclined to begin their implementations prematurely—a process whose comparatively ample feedback may mask underlying misunderstandings and instill a false sense of progress.

In this dissertation, I demonstrate that providing students with timely feedback on their functional examples incentivizes them to develop functional examples, improves the quality of their test cases, and may improve the correctness of their implementations.
Executable Examples
Empowering Students to Hone Their Problem Comprehension

by
John Sinclair Wrenn
B. A., University of Rhode Island, 2015
Sc. M., Brown University, 2018

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
April 2022
This dissertation by John Sinclair Wrenn is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________  ______________________________________

Shriram Krishnamurthi, Director

Recommended to the Graduate Council

Date ______________  ______________________________________

Kathi Fisler, Reader
Brown University

Date ______________  ______________________________________

Mark Guzdial, Reader
University of Michigan

Date ______________  ______________________________________

Tim Nelson, Reader
Brown University

Date ______________  ______________________________________

Joe Politz, Reader
University of California, San Diego

Approved by the Graduate Council

Date ______________  ______________________________________

Dean of the Graduate School
ACKNOWLEDGEMENTS

To my parents, for everything, always.

To Asha, without whom I would not have lasted a month.

To Joshua, Zoe, Peter, and Scott and the Inning Club, for revelling with me in Brown University’s highs and lows.

To Lionel, for all the wry smiles and dirty jokes.

To Rebecca, for crowning me King of Wheat.

To Zoe, for seeing me across this finish line and beyond.

To Donna and Alex, for the black raspberry chocolate chip frozen yogurt, and so much more.

To my mentors, especially:

    Shriram, who taught me to research, write, and speak.
    Tim and Kathi, who taught me how to survive academia.
    Deborah, who sparked my love for computing education.

To Max, without whom Examplar would not have seen daylight in csci0190.

To the NSF, for partially funding this work.
# CONTENTS

List of Tables xi  
List of Figures xiii  

## 1 Introduction  
1.1 Self-regulation through systematic problem solving 2  
1.2 Supporting systematic problem solving 4  
1.3 Executable Examples 5  

## 2 Assessing Input–Output Examples 9  
2.1 Measures 9  
  2.1.1 Relationship Between Metrics 10  
  2.1.2 Relationship to Binary Classification Metrics 11  
  2.1.3 Curating Wheats and Chaffs 11  
2.2 Theoretical Limitations 12  
  2.2.1 When correctness is not binary. 12  
  2.2.2 When wheats, chaffs behave nondeterministically. 13  
  2.2.3 When checking whether an input is within the domain is undecidable. 13  
  2.2.4 When chaffs might not terminate. 13  
  2.2.5 When there is no well-defined API. 14  
  2.2.6 When examples are hard to write. 14  
2.3 Alternatives 15  
  2.3.1 Coverage 15  
  2.3.2 Mutation Testing 16  
2.4 Conclusion 16  

## 3 Examplar-A: An IDE for Executable-Examples 19  
3.1 Execution & Feedback Model 20  
  3.1.1 Assessing Validity 21  
  3.1.2 Assessing Thoroughness 22  
3.2 Assignment Model 22  

## 4 Other Ways to Incentivize Early Testing 25  
4.1 Peer Review 25  
  4.1.1 Code Defenders 25  
  4.1.2 Captain Teach 25  
4.2 Testing Quizes 26  
  4.2.1 Prather et al. 26  
4.3 External On-Demand Automated Feedback 26
A.1 Favor covering the API 105
A.2 Favor logical errors over programming errors 106
A.3 Avoid chaffs with multiple bugs 106
A.4 Avoid difficult-to-catch chaffs 107
A.5 Avoid long-running wheats & chaffs 107
A.6 Avoid non-deterministic chaffs 108
A.7 Never use chaffs prone to non-termination. 108
A.8 Never use ill-typed chaffs 109

Bibliography 111

* Parts of this dissertation were adapted from prior publications of the author, namely [1] [2] [3]. These appearances are noted in the titlenotes of chapters adapting their prose.
#### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>The ordinal of assignments between offerings of csci0190. An assignment is listed as having multiple ordinals if it was broken up into sub-assignments prior to the Fall 2020 offering of csci0190.</td>
</tr>
<tr>
<td>Table 2</td>
<td>The length of assignments between offerings of csci0190, in days. The lengths of projects divided into multiple assignments are summed together.</td>
</tr>
<tr>
<td>Table 3</td>
<td>The position and duration of the comparable assignments in each year.</td>
</tr>
<tr>
<td>Table 4</td>
<td><em>Did you use Examplar</em> (\alpha) <em>to write examples or tests Before, During, and After completing your implementation?</em> (Higher percentages are shaded darker.)</td>
</tr>
<tr>
<td>Table 5</td>
<td>For each of the comparable assignments and in aggregate: the proportion of the n final test suites which were invalid.</td>
</tr>
<tr>
<td>Table 6</td>
<td>For each of the comparable assignments and in aggregate: the proportion of final test suites of each form of invalidity: (1) accepting <em>Some</em>—but not all—wheats, (2) accepting <em>None</em> of the wheats because one or more test cases failed, and (3) accepting none of the wheats because an <em>Error</em> prevented the suite from running.</td>
</tr>
<tr>
<td>Table 7</td>
<td>For each of the comparable assignments and in aggregate: the number of chaffs used by Examplar(\alpha), the Final number of chaffs used to assess students’ final test suites, and the proportion of Final chaffs caught, on average, by students’ final test suites.</td>
</tr>
<tr>
<td>Table 8</td>
<td>For each of the comparable assignments and in aggregate: the proportion of the n final implementation submissions for each year that were correct.</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Examplar$_{\alpha}$, the first of two versions of our specialized IDE for example-writing.</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Examplar$_{\alpha}$ provides a specialized editing environment for writing examples. Run Tests assesses the quality of the suite by running it against wheats and chaffs. The suite above is valid (i.e., it accepts both wheats), but it is not particularly thorough (it rejects only two of the four included chaffs); its assertions are also valid for a different summary statistic: mean!</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Upon each click of “Run”, Examplar$_{\alpha}$ successively runs the students’ tests against instructor-authored wheat and chaff implementations.</td>
</tr>
<tr>
<td>Figure 4</td>
<td>While the wheats are sequentially executed, Examplar$_{\alpha}$ tallies which tests failed (and are thus invalid). If any invalid tests are detected, the invalid tests are reported to the student (without revealing why the tests failed), and thoroughness is not subsequently assessed.</td>
</tr>
<tr>
<td>Figure 5</td>
<td>If an unexpected error halts any check blocks in the tests file from executing, the execution of wheats ceases and the invalidity is reported.</td>
</tr>
<tr>
<td>Figure 6</td>
<td>If all tests are valid, Examplar$<em>{\alpha}$ then assesses the thoroughness of the suite by running its tests against each chaff. In Examplar$</em>{\alpha}$’s feedback, chaffs are represented with bug icons, which are shaded blue when the chaff is caught. Mousing over the chaffs highlights the tests that rejected it.</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Each Examplar$_{\alpha}$ assignment is backed by an instructor-authored, public Google Drive folder that contains wheats, chaffs, and a template for the students’ tests.txt file.</td>
</tr>
</tbody>
</table>
For each assignment: a combined violin and box-and-whiskers plot illustrating the volume of Examplar\textsubscript{α}-submissions-per-student for each assignment. The center line in each box represents the number of times the median student clicked Run Tests. The whiskers extend to the most extreme data lying within 1.5 times the interquartile range. Points indicate the exact number of submissions of outlying students.

Examplar\textsubscript{α} submissions per-student for each of DataScripting parts, rendered in the same manner as fig. 8.

The number of assertions in final test suites from each year, rendered in the same manner as fig. 8.

Examplar\textsubscript{α} received attention on a university Facebook page for anonymous admiration.

Examplar\textsubscript{β} provides a unified editing environment for example-writing, implementation, and testing. Once the student begins their implementation, Examplar\textsubscript{β} executes the student’s tests against their implementation. The above student has conflated median for mean in their implementation, but not in their test cases. Accordingly, Examplar\textsubscript{β} informs them that their tests are valid, but that one of these valid test cases has rejected the student’s own implementation. The failure of a valid test implies a bug in the student’s implementation.

As with Examplar\textsubscript{α} assignments, each Examplar\textsubscript{β} assignment is backed by an instructor-authored, public Google Drive folder that contains wheats, chaffs, and a template for the students’ tests\texttt{.arr} file. For Examplar\textsubscript{β} assignments, this folder must also contain a “dummy implementation” (so the repl works before the student has begun their implementation), and (optionally) templates for students’ code and common files.

Upon each click of “Run”, Examplar\textsubscript{β} successively runs the student’s tests against instructor-authored wheat and chaff implementations. If the student’s tests are valid, Examplar\textsubscript{β} additionally runs the student’s tests against their own implementation.
While the wheats are sequentially executed, Examplarβ tallies which tests failed (and are thus invalid). If any invalid tests are detected, the invalid tests are reported to the student (without revealing why the tests failed), and thoroughness is not subsequently assessed.  

If an unexpected error halts any check blocks in the tests file from executing, the execution of wheats ceases and the invalidity is reported.  

If all tests are valid, Examplarβ then assesses the thoroughness of the suite by running its tests against each chaff. In Examplarβ’s feedback, chaffs are represented with bug icons, which are shaded blue when the chaff is caught. Mousing over the chaffs highlights the tests that rejected it.  

Examplarβ discourages using the REPL discover the correct output of functions. Prior to the clicking of Begin Implementation, REPL executions occur in the context of a “dummy implementation”—a wheat in which every function body has been stubbed out with raise('output hidden').  

Each point reflects the relative testing and implementation effort of a student, measured in standard deviations from their typical testing and implementation efforts. Implementation effort ranges on the x-axes; testing ranges on the y-axes. Assignments are parenthetically numbered with the index of their appearance in the course.  

The misfit of each student (represented by points) on each assignment. The shaded areas with fractional labels reflect each possible level of thoroughness as the proportion of buggy implementations rejected. Underlaid box-and-whiskers plots summarize the overall misfit on each assignment.
INTRODUCTION

The computing education literature is rife with studies in which student participants inadvertently make significant progress solving the “wrong” problems. Whalley and Kasto [4], for instance, conducted a think-aloud study of novice programmers tasked with solving problems in a graphical, Karel-the-Robot-esque “robot world”, in which students programmed a character to move around randomly generated hallways. Students prematurely retrieved a plan and applied it, not realizing it was inadequate; the interviewer had to prod the students before they realized they had made an error at all:

Interestingly, [three of six students] retrieved the ‘counting integers’ schema. The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

Similarly, Prather et al. [5] conducted a think-aloud study of students attempting to solve a programming problem in the presence of an automatic assessment system that would run students’ code against various test cases. The authors hypothesized that access to this on-demand feedback would help students navigate the implementation phase of the problem-solving process. Instead:

The most frequent issue these students encountered was a failure to build a correct conceptual model of the problem.

Consequently, they tended to apply familiar-yet-inadequate plans, and entered a myopic cycle of encountering and fixing errors, possibly exacerbated by the presence of automated feedback:

The feedback from Athene seems to have given [several participants] a false sense of progression through the problem.

This chapter adapts content previously published by the author in Executable Examples for Programming: Problem Comprehension [11].
Analyzing the programming process of 37 students across two experience groups (CS1 and CS2 students), Loksa and Ko observed:

Participants often began coding without fully understanding the problem, leaving them with knowledge gaps in the problem requirements and causing them to later stop implementation to address the gaps.

Amidst a field focused largely on the task of teaching students how to program, these think-alouds suggest a vital complimentary skill: knowing when to program. Loksa and Ko hypothesized that this failure occurred because novices lack the metacognitive awareness to self-regulate their progress through the problem-solving process. While educators vary slightly on what, exactly, this process consists of, generally speaking, it is a process that begins with understanding the problem, and ends with reviewing one’s solution. This critical, early stage of building problem understanding occurs through *reinterpretation* — the act of rephrasing and reformulating a problem statement to build understanding of it. Only a minority of their participants (15 of 37) vocalized this stage of the problem solving process.

1.1 Self-regulation through systematic problem solving

One potential way to improve students’ development of problem comprehension is to train them in a methodology that explicitly scaffolds the process of problem solving. Educators have long attempted to support metacognition by instructing students in an explicit problem solving methodology, ranging from Pólya’s 1945 *How to Solve It* [7] for mathematics, to *How to Design Programs*’s “Design Recipe” [8], to Loksa et. al’s 2016 six-step metacognitive scaffold [9]. All three of these scaffolds ask students to begin by *reinterpretating* the problem to ensure they understand it, and to end by reviewing their solution.

George Pólya’s 1945 manual *How to Solve It* [7] advocates novice mathematicians follow a four-step problem solving process:

1. **Understand the problem.** “You have to *understand* the problem.”
   Pólya instructs students to first consider the unknown, identify the data, and the condition that must be satisfied. He recommends students explicitly reinterpret the problem drawing a figure, selecting a suitable notation, and writing down the sub-parts of the condition to be satisfied.

2. **Devise a plan.** “Find the connection between the data and the unknown.”
   Pólya instructs students to question whether they have seen the problem before, or seen similar problems, and adapt their solutions. Failing that, he urges students to try to restate the problem, or else attempt to first solve an easier variant of it.
3. **Carrying out the plan.** “Carry out your plan.”

Pólya instructs students to decompose their plan into steps, execute each step, and prove that each step is correct.

4. **Looking back.** “Examine the solution obtained.”

Finally, Pólya instructs students to check the result, the argument, and examine whether the result could be derived differently.

In the pedagogical contexts considered within this dissertation, students are instructed in the Design Recipe from *How to Design Programs* (htdp) [10], a six-step process for producing an implementation from a specification:

1. **From Problem Analysis to Data Definitions**
   - Identify what must be represented and how it is represented.

2. **Signature, Purpose Statement, Header**
   - State what kind of data the function consumes and produces.

3. **Input–Output Examples**
   - Work through examples that illustrate the function's purpose.

4. **Function Template**
   - Translate the data definitions into an outline of the function.

5. **Function Definition**
   - Fill in the gaps in the function template.

6. **Testing**
   - Ensure your implementation conforms to your examples.

The first three of these steps specifically scaffold the development of problem comprehension, and its last step prompts students to confirm that their understanding matches their implementation.

At a high level, its steps provide a form of scaffolding [11] to lead a student from a prose-based problem statement to a working program. The scaffolding steps ask students to produce intermediate artifacts (signature/purpose, examples, code template) that capture the problem at multiple levels of detail and abstraction. The progression from data definitions to examples to code move the student through different representations of the problem, providing a form of concreteness fading [12] as students progress towards a symbolic-form solution to a problem.

Completed sequences of design-recipe steps form worked examples [13] that students can leverage when considering new problems. A student might refer to a design recipe example when writing a new program on an already-studied datatype: this would focus on the examples, templates, and code features of the example. When asked to work with a new datatype, the recipe suggests higher-level steps that a student can follow to make progress on the problem.
Templates are a form of program schema [14][15] that students can recall and reuse in constructing solutions to new problems. The LISP Tutor [16] builds on a theory that students can recognize and adapt solutions to recursive problems, though without an explicit step of articulating the template independently from the code. The template, in contrast, provides an explicit scaffold that handles traversing an entire data structure as part of implementing a solution to a specific problem.

Several papers have begun to explore the positive impact of the HTDP recipe on students in different contexts. Fisler and colleagues on multiple projects [17][18] showed that HTDP-trained students made fewer programming errors than students trained in more conventional curricula. Schanzer et al. [19][20] have found improvements in middle- and high-school students’ abilities to solve algebra word problems after working with a version of the design recipe.

1.2 SUPPORTING SYSTEMATIC PROBLEM SOLVING

Of course, students trained to follow HTDP might, nonetheless, not follow the design recipe. For instance, Fisler et al. [21] conducted a think-aloud study highlighting the experience of four students tasked with implementing Soloway’s Rainfall problem. Although these students were encouraged to follow the design recipe, none formulated any examples, and subsequently struggled with the problem. Similarly, a study by Edwards and Shams [22] of students trained in test-driven development and graded on test suite coverage found that students’ tests were both few and uninformative; most students wrote only exactly as many tests as there were methods, and those tests tended to only evaluate the “happy path” of their respective methods. Such mishaps illustrate the crux of the self-regulation conundrum: although instructors can teach students techniques that encourage self-regulation, choosing to practice those techniques is, itself, a feat of self-regulation!

Consequently, we might advocate that instructors merely require that students write examples before beginning programming [23][24]. Instead, we ask: Are there any reasons why example-writing might be less compelling to students than we think it ought to be?

Yes. For one, input–output examples are completely inanimate until the student completes their implementation (HTDP step 5), at which point they become the basis of a test suite (HTDP step 6). In short: you cannot run your input–output examples until you have an implementation to run them against. An implementation, on the other hand, may be run almost as soon as it is begun. Prather et. al [5] hypothesize that students may begin their implementations prematurely (at the expense of problem comprehension) because the implementation phase
of problem solving provides interactive feedback (albeit counter-productive); comprehension development does not.

We also observe that writing input–output examples cannot dispel consistent misconceptions about the problem—even once those examples are adapted into test cases. If a students’ misconception about a programming problem is consistently reflected in both their examples and implementation, running those examples as test cases will fail to detect any error.

Unalloyed, input-output examples are neither compelling, nor possibly even helpful. There is no mechanism within HtDP by which students can get timely feedback as they write input–output examples, or assess whether those examples conform to the problem specification. This dissertation proposes such a mechanism: executable examples.

### Thesis

Providing students with timely feedback of their input–output examples incentivizes them to develop input–output examples, improves the quality of their test cases, and may improve the quality of their implementations.

1.3 EXECUTABLE EXAMPLES

To make input–output examples animate, we execute them—though not against students’ own implementations (which, in principle, may not yet exist). Rather, we apply an assessment model (detailed in chapter 2) that can provide feedback on their quality without the student having begun their implementation. We first implement this model in Examplar_α (pictured in fig. 1 detailed in chapter 7), a development environment standalone from students’ usual implementation and testing environment, which is specialized for writing examples. We primarily deploy Examplar_α in a college-level accelerated introduction to computer science course (henceforth: csci0190). The course, its assignments, and scope of Examplar’s deployment is detailed in chapter 5.

The remainder of this dissertation builds on this foundation. Chapter 6 directly addresses the thesis of this dissertation: it describes the deployment of Examplar_α to csci0190, and reports on a study demonstrating that Examplar_α was appealing, helpful, and seemingly non-harmful. Use of Examplar_α was not required; students nonetheless used it substantially. The quality of students’ final test suites improved significantly over the previous offering of the course.

The promising results of this study prompted us to more deeply integrate executable example feedback into students’ workflows. In chapter 7 we describe the successor to Examplar_α, “Examplar_β”, which integrates executable example
feedback into the same development environment students use for testing and implementation. This integration, in principle, reduces the self-regulation required from students to make use of executable example feedback: students no longer need to consciously mediate their time between two separate development environments.

The integration also gave us insight into how students allocated their effort between example-writing, testing and implementation. Chapter 8 describes and applies a novel measure for characterizing how thoroughly students explored their homework problems with examples before undertaking the bulk of their implementation work. Although this study was inadequate for attributing students’ behavior to the presence of Examplarα (a before-and-after comparison in the style of chapter 8 was not possible), we resoundingly did not observe any widespread “test-last” behavior from students; students largely did write examples before the bulk of their implementation work.

A student’s understanding of a problem does not only impact their interactions with their code, it also influences their help-seeking interactions with course staff. In chapter 9 we conduct a thematic analysis of the questions students asked regarding the input–output behavior of assignments to csci0190’s online course forum. We find that many questions were prompted by Examplarβ’s feedback. This reflects on Examplarβ’s value in the course—it prompted students to pose questions to the course staff rather than proceed unawares with a misconception—but also its shortcomings: Examplarβ’s feedback was not so elucidating as to obviate the role of course staff. We also observe many questions that could have been answered by Examplarβ, had the student first posed their question as an input–output example; this suggests a degree of latent utility.

Although csci0190 was the primary context in which we studied Examplar, it was not the only context we deployed the tool in. We deployed Examplarα in a
programming language implementation and design course, and Examplarβ in a segment of a “relaxed” introductory course (csci0111). Neither of these deployments was as smooth as that of csci0190. The deployment into the PL course was complicated by limitations of the assessment model; these complications are described in section 2.2.4. The utility of Examplarβ in csci0111 was stymied by pedagogic factors; we detail these obstacles in chapter 10.
ASSESSING INPUT–OUTPUT EXAMPLES

Input–output examples, like test cases, can be articulated as assertions. To assess whether examples are both valid and thorough explorations of a problem, we adapt the classifier perspective of assessing test suites [26, 27]. This perspective views suites as classifiers of implementations, judging them as either correct or buggy. Consequently, one can assess the quality of a test suite by seeing how accurately it classifies sets of known-buggy and known-correct implementations. A good test suite will mislabel few correct implementations, and catch most buggy implementations.

However, unlike test cases, the intent of examples is not to test one’s implementation, but rather one’s understanding of the problem; we adapt our application of this perspective to reflect this difference (see section 2.1.3).

2.1 MEASURES

We adopt the measures of validity and thoroughness to quantify the quality of test suites.

VALIDITY Validity is a binary measure. A suite is valid if it accepts (i.e., its assertions pass) all correct implementations. Otherwise, it is invalid. A suite may be invalid for a variety of reasons; particularly, it may have:

1. asserted incorrect behavior (e.g., sorting in the wrong direction),
2. asserted underspecified behavior (e.g., asserting that a sorting implementation is stable, if that was not specified),
3. simply have failed to compile or run altogether.

We assess whether a test suite is valid by running it against instructor-authored, known-correct implementations (henceforth wheats [26]).

1 This chapter adapts content previously published by the author in Who Tests the Testers? [25], Executable Examples for Programming Problem Comprehension [11], and Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors [3].
THOROUGHNESS A suite is thorough if it rejects (i.e., its assertions do not pass) buggy implementations. We assess the thoroughness of a suite by running it against a curated set of buggy implementations (henceforth chaffs [26]). The thoroughness of a suite is measured as the proportion of chaffs it rejects. In section 2.1.3 we discuss our curation of wheats.

2.1.1 Relationship Between Metrics

Validity and thoroughness are jointly important for assessing the quality of a test suite. It is trivial to construct a test suite that rejects all buggy implementations; e.g.:

```plaintext
check:
  1 is 2
end
```

...but such a test suite also rejects all correct implementations; it is not of any practical value. Since the bug-catching capabilities of a test suite must not come at the cost of rejecting correct behavior, the thoroughness of a test suite is predicated on its validity.

Our application of these metrics reflects this dependency. First, we assess validity and identify any invalid tests. Then, before assessing thoroughness, we must (at least) discard these invalid tests. The auto-grading system of csci0190 performs this filtering automatically. Examplar, in contrast, highlights the students’ invalid tests and requires that they are corrected (or eliminated) before thoroughness is assessed. We describe Examplar’s behavior more fully in chapters 3 and 7.

In both cases, validity is treated as the lowest bar a student must clear for their test suite to be eligible for further assessment. A test suite that achieves validity but then fails to demonstrate a modicum of thoroughness is nearly as useless as the inverse — it is equally trivial to construct a test suite that accepts all possible correct (and buggy) implementations; e.g.:

```plaintext
check:
  1 is 1
end
```

Despite achieving perfect validity, clearly this is not a ‘half-credit’ test suite.

Those who would give up essential Validity, to purchase Thoroughness, deserve neither Validity nor Thoroughness.
2.1.2 Relationship to Binary Classification Metrics

These metrics are closely related to two of the conventional binary classification metrics. The validity metric is closely analogous to the conventional metric of true-negative rate, the proportion of classifications where both the true and detected conditions are negative. Conversely, the thoroughness metric is closely analogous to the conventional metric of true-positive rate, the proportion of classifications where both the true and detected conditions are positive. In this dissertation, we prefer the terms “validity” and “thoroughness” for both social and semantic reasons:

1. “Positive” is often colloquially associated with desirable conditions, and “negative” with undesirable conditions. In the context of software testing, the desired condition typically involves passing tests. This association is opposite of the technical use of positive and negative in the context binary classifiers. Just as medical tests detect illness, not wellness, software tests detect bugginess, not correctness. However, rather than subjecting our peers to this pedantry, we instead select terms whose value connotations match their technical meaning: it is “good”, both colloquially and technically, for a test suite to be valid and thorough.

2. We do not assess these metrics independently. As previously described, we do not permit invalid tests to contribute to validity.

2.1.3 Curating Wheats and Chaffs

Wheats If the problem specification leaves any behavior underspecified, it is necessary to run suites against multiple correct implementations in order to accurately identify invalidity. For instance, consider a problem specification that reads:

Write a function, median, that consumes a list of numbers and produces the arithmetic median.

This specification, as worded, leaves the behavior of median on empty inputs underspecified; it may be just as correct for an implementation to produce an error as to return 0. In order for a suite to be valid for all implementations of median, it must not include any assertions involving empty input lists. We can accurately identify such assertions as invalid by checking them against two correct implementations:

1. one that produces an error on empty inputs,
2. another that produces some answer (say $\theta$) on empty inputs.

If a student asserts that implementations should produce an error on empty inputs, their suite will reject the wheat that produces $\theta$ (and visa versa). Provided that the set of wheats completely exercises the space of underspecified behaviors permitted by the specification, accepting all wheats guarantees that a suite is valid and will accept all correct implementations. (This is not always possible, as discussed in section 2.2.)

**CHAFFS** To comprehensively assess how good a test suite is at detecting bugs, the set of chaffs should include a wide variety of faults, both subtle and major. The set of chaffs used for grading students’ final submissions on each csci0190 assignment includes upwards of 20 buggy implementations ranging widely in severity and subtlety. In Examplar, by contrast, we want to assess how thoroughly have explored the conceptually interesting aspects of the problem—not how good the student is at catching off-by-one errors. The selection of chaffs for Examplar should reflect this difference, exercising logical misunderstandings that students are likely to make. For instance, to assess the thoroughness of examples for median, the set of chaffs could include implementations of mean and mode. The course staff of csci0190 applied this principle to select chaffs for Examplar from the superset of chaffs used for final grading. For further guidance (much of which stems from hard lessons in the semesters since we first deployed Examplar), see appendix A.

### 2.2 Theoretical Limitations

There are a handful of situations in which Examplar is poorly suited:

#### 2.2.1 When correctness is not binary.

In the classifier approach to test suite assessment, correctness must be a well-defined, binary property of implementations. We could not provide Examplar for two assignments which lacked this property. For instance, in Sortacle, students implemented the function sortacle, which consumed a sorting function and produced true if the function was correct (and false, otherwise) by checking it against a large number of generated inputs. A quality sortacle will be very good at labeling sorts accurately, but this is an impossible task to do perfectly: it is always possible to craft a sorting function so deviously buggy that no sortacle will detect the flaw. Since it is impossible to craft a true wheat for such an assignment, the classifier approach is inappropriate.
2.2.2 When wheats, chaffs behave nondeterministically.

Examplar can be used for assignments that allow for an element of non-determinism, a form of underspecified behavior. However, if the wheats and chaffs loaded into Examplar are non-deterministic between runs, Examplar may provide students with differing feedback between runs of the same suite. The allowance of non-determinism should therefore be realized by loading Examplar with multiple wheats that differ in their behavior, but are individually deterministic.

2.2.3 When checking whether an input is within the domain is undecidable.

To detect tests of unspecified behavior as invalid, instructors must craft multiple wheat implementations that observably differ in their unspecified behavior. For instance, a function that is defined for non-empty input lists suggests two wheats that handle the empty-input differently: one that produces a value on an empty input, and another that produces an error. This approach to developing wheats is sufficient for problems in which it is possible to efficiently decide whether the function’s input is outside of its specified domain.

However, if the procedure-under-test consumes a function as an argument — one whose validity depends on it satisfying a behavioral invariant — it may be impossible to efficiently detect (and, consequently, respond to) if the input is out-of-domain.

For instance, the JoinLists assignment of csci0190 asks students to implement j-sort,

\[
\text{j-sort}(A)(\text{cmp} :: (A, A \rightarrow \text{Boolean}), \text{jl :: JoinList}(A)) \rightarrow \text{JoinList}(A)
\]

where \text{cmp} is a comparator predicate that’s satisfied if its first argument is greater than its second argument. If \text{cmp} is “buggy” (e.g., it does not actually define a valid ordering of \text{A}), the behavior of \text{j-sort} is unspecified. Thus, to detect invalid tests of \text{j-sort}, the instructor must provide wheats that observably differ when a buggy \text{cmp} is provided. Unfortunately, proving or disproving the correctness of an arbitrary \text{cmp} is undecidable. Wheats are inadequate for detecting this sort of invalidity.

2.2.4 When chaffs might not terminate.

Examplar’s feedback is provided upon each click of Run, within the flow of development. Care should be taken to ensure that running the wheats and chaffs does not take too long. A particularly pernicious threat to timely feedback are
chaffs that are buggy in ways that could lead to non-termination. This is a threat unique to Examplar, not the assessment model as a whole.

In the Fall 2018 offering of csci1730: Programming Languages\(^3\), students implemented a series of program interpreters; accordingly, the chaffs were comprised of buggy interpreters. In a correct implementation of these interpreters, a program like:

\[
(\text{let } [f \text{ (lambda (x) (f x))}] \\
(f 0))
\]

is specified to produce an unbound identifier error. The wheats terminate with this error virtually immediately. In contrast, a buggy chaff that implements `let` as `let-rec` will recur infinitely.

In final grading, this had not posed an issue: the auto-grader timed-out chaffs runs at ten minutes, and counted timeouts (or running out of memory) as catching the chaff. Unfortunately Examplar strongly disincentivizes such tests: a non-terminating chaff will, given enough time, crash one’s browser tab. This pitfall forced students to comment-out or delete such thorough tests entirely in order to make forward progress on the assignment.

2.2.5 *When there is no well-defined API.*

Our assessment model requires that the instructor author sets of correct and buggy implementations that match the API of the students’ implementation. Our assessment model is thus only suitable for use on assignments where a public API is defined.

2.2.6 *When examples are hard to write.*

Examplar is not useful when articulating examples accurately is difficult or impossible. For instance, TourGuide asked students to implement a function that consumes a graph of locations, start points and end points, and produces the length of the shortest path between those termini. Unfortunately, the Euclidian distances between points is very often *irrational* and therefore must be represented approximately in floating point. To complicate matters, the accuracy of a summation of floating point numbers depends on the order in which the numbers are added. Thus, it is very difficult to express the right answer accurately on a computer.

\(^3\) [http://cs.brown.edu/courses/csci1730/2018/](http://cs.brown.edu/courses/csci1730/2018/)
2.3 Alternatives

Assessing the validity of student tests by ensuring it accepts an instructor-authored implementation is commonplace [28, 22, 20, 27, 30, 26]. This check is particularly important when using student tests to assess each others’ implementations [28, 22, 20, 31].

Similarly, many instructors assess the quality of student tests as we do, by running them against a corpus of incorrect implementations, and checking what fraction of these a test suite rejects. This corpus may be sourced from students [12, 33, 34, 22, 31], from machine-generated mutations of a reference implementation [35, 36, 34], or—as in our case—crafted by the instructor [26, 27]. Alternatively, instructors can use code coverage as a proxy for thoroughness.

2.3.1 Coverage

Coverage measures quantify the ‘amount’ of code of an implementation that is explored in the course of running tests. Statement coverage, for instance, counts the proportion of syntactic statements executed at any point in the course of running tests; a variation, edge coverage computes the proportion of edges in the program control flow graph that have been explored. Coverage metrics are efficient to compute, are available in many popular programming languages and development environments, and do not require the presence of a separate reference implementation (and, consequently, is suitable on assignments with no well-defined API). Because of these qualities, coverage metrics are popular in professional software engineering, lending authenticity to their use in classroom settings. [37] The automated assessment systems ASSYST [38], Web-CAT [28], and Marmoset [39] all use coverage.

However, coverage is not truly a measure of how well a test suite explores a problem, merely how well it explores an implementation. A student achieve both perfect validity and perfect coverage by making many method calls with varied inputs, all-the-while not asserting any properties about their outputs. [27] Such a suite will likely “cover” any given implementation, but utterly fails to be an alternative representation of the problem specification.

Furthermore, although coverage-based metrics do not require the presence of an instructor-authored reference implementation, they nonetheless require that some implementation is available against which tests can be executed. When this implementation is not one provided by the instructor, it is typically the student’s own implementation that coverage is assessed on. In such circumstances, obtaining meaningful feedback from coverage requires first attaining a reasonably complete
implementation; such an arrangement clearly fails to encourage an examples-first development style.

Perhaps out of some combination of these factors (and more), empirical studies of code coverage find that it is a poor proxy for thoroughness. A study by Edwards and Shams [22] of students trained in test-driven development and graded on test suite coverage found that students’ tests were both few and uninformative; most students wrote exactly as many tests as there were methods, and those tests tended to only evaluate the “happy path” of their respective methods.

2.3.2 Mutation Testing

In the mutation testing [40] approach to test suite assessment, a given, base implementation (either instructor’s or the student’s own implementation) is programmatically “mutated” into buggy variations. Mutation testing is an effective technique by which software engineers can assess the efficacy with which their test suites will detect real-world faults [41]. As arbitrarily large sets of mutants can be generated very cheaply, this method is seemingly an attractive alternative to manually-authoring chaffs.

However, the unsupervised generation of mutants poses its own challenges in pedagogic contexts:

1. The distribution of bugs introduced by random mutations is likely to be dissimilar from the distribution of bugs desired by instructors for pedagogic reasons. Chaffs used for assessing student understanding, for instance, should exercise the conceptually interesting corners of problems—(usually) not off-by-one errors.

2. While a variety of methods exist for generating mutants, these do not generally also produce human-readable explanations of the bugs they contain. When a test suite fails to reject a mutant, a programmer (either the instructor or student) must manually decipher the mutation to produce an actionable plan for improving the suite.

3. The mutant-generating process may create mutants that fail to introduce bugs (the “equivalent mutant problem”). These non-buggy mutants must be accounted for, but doing so is difficult [42].

2.4 Conclusion

The range of assignments our assessment model can be used on is more limited than that of either mutation testing or coverage-based metrics, both of which can
be used on assignments where no instructor-authored reference implementation exists.

However, this apparent advantage of the alternatives comes at a cost: when the student’s own implementation is used to compute coverage or as a basis for mutants, the usefulness of the resulting measures depends on the completeness of the student’s implementation. So, even if we were to adopt coverage or mutation testing for Examplar, we would still need to produce a reference implementation so that feedback could be provided before implementation. We are, then, still limited to assignments with well-defined APIs. And, once a reference implementation has been carefully authored by the instructor, it is relatively easy to hand-mutate it into additional wheats and chaffs.

This artisinal approach to assessing validity affords the instructor a degree of control mutants lack: the set of chaffs can be curated to excercise whatever qualities of the problem the instructor feels are pedagogically important. We take advantage of this control in Examplar: whereas chaffs used for final grading exercise subtle implementation bugs, chaffs selected for examplar can correspond to major, logical misconceptions of the problem.
EXAMPLAR$_A$: AN IDE FOR EXECUTABLE-EXAMPLES

Figure 2: Examplar$_A$ provides a specialized editing environment for writing examples. Run Tests assesses the quality of the suite by running it against wheats and chaffs. The suite above is valid (i.e., it accepts both wheats), but it is not particularly thorough (it rejects only two of the four included chaffs); its assertions are also valid for a different summary statistic: mean!

Examplar$_A^2$ (pictured in fig. 2) provides a specialized version of the usual Pyret$_4$ editing environment$_3$ tuned for writing examples as input-output assertions. Students write their assertions just as they would in Pyret’s usual editor. However, Examplar$_A$ replaces the usual editor’s Run button with a Run Tests button, which assesses the student’s suite for validity and thoroughness against instructor-authored implementations (in the manner described in section 2.1). Consequently, students can use Examplar$_A$ to develop and assess their examples independent of their implementation progress.

---

1 This chapter adapts content previously published by the author in Executable Examples for Programming Problem Comprehension [1].
2 https://github.com/brownplt/examplar
3 https://code.pyret.org/editor
Examplar, in contrast to the usual Pyret IDE (CPO), is an IDE expressly for the development of examples. Whereas the CPO provides an editor and execution environment for any Pyret file (be it code.arr, tests.arr or common.arr) Examplar only provides an editing environment for an assignment’s tests.arr file. CPO’s “Run” button is replaced in Examplar with a Run Tests button, which initiates the example quality feedback process. Examplar does not ever execute the student’s tests.arr file against their own code.arr file.

Instead, upon each click of “Run Tests”, Examplar successively runs the students’ tests against instructor-authored wheat and chaff implementations, and must specially handles a variety of success and failure modes of Pyret test execution. First, Examplar runs the student’s tests against the wheat implementations to determine whether the tests are valid. Then, it runs the student’s tests against the chaff implementations. Figure 3 illustrates Examplar’s evaluation model.

4 These are the three files submitted by students on most assignments in csci0190. For more information about these deliverables, see section 5.2.
include my-gdrive("median-code.arr")
# DO NOT CHANGE ANYTHING ABOVE THIS LINE
check:
  median([list: 1]) is 1
  median([list: 1, 2]) is 1.5
  median([list:]) is 0
  median([list: 1, 2, 3]) is 2
  median([list: 0, 0, 1, 2, 3]) is 1
end

Your tests rejected 1 out of 2 wheats:

The wheats your tests rejected are highlighted above in red. Mouseover a wheat to see which of your tests rejected it. Are these tests consistent with the problem specification? Do they test unspecified behavior?

Figure 4: While the wheats are sequentially executed, Examplar$_a$ tallies which tests failed (and are thus invalid). If any invalid tests are detected, the invalid tests are reported to the student (without revealing why the tests failed), and thoroughness is not subsequently assessed.

Figure 5: If an unexpected error halts any check blocks in the tests file from executing, the execution of wheats ceases and the invalidity is reported.

### 3.1.1 Assessing Validity

Examplar$_a$ begins by executing the students’ test suite against each of the wheat implementations. While the wheats are sequentially injected, Examplar$_a$ tallies which, if any, of the students’ tests have failed (and are thus invalid). If any invalid tests are present after wheat execution completes, those invalid tests are reported to the student without revealing why the tests failed (pictured in fig. 4) and thoroughness is not subsequently assessed.

If a wheat fails because an unexpected error was encountered in a check block, the invalidity is reported (as in fig. 5) and the further execution of wheats ceases.
Figure 6: If all tests are valid, Examplar_{\alpha} then assesses the thoroughness of the suite by running its tests against each chaff. In Examplar_{\alpha}’s feedback, chaffs are represented with bug icons, which are shaded blue when the chaff is caught. Mousing over the chaffs highlights the tests that rejected it.

CPO provides extensive information in its presentation of errors and testing results\textsuperscript{5}. For instance, if a test fails because the two halves of an equality assertion are not equal, CPO displays the values that each half evaluated to. This is undesirable in Examplar_{\alpha}, as a student may be overly tempted to intentionally write failing assertions to discover what the behavior of wheats is, rather than closely read the assignment specification to determine the behavior on their own. Our intention is that Examplar_{\alpha} supplements—but does not replace—the assignment specification. Examplar_{\alpha} therefore removes the interaction pane and suppresses nearly all forms of program output. Examplar_{\alpha} only displays errors that prevent assertions from running.

3.1.2 Assessing Thoroughness

If all tests are valid, Examplar_{\alpha} then assesses the thoroughness of the suite by running its tests against each chaff. As shown in fig. 6, chaffs are represented with bug icons, which are shaded blue when the chaff is caught; mousing over the chaffs highlights the tests that rejected it.

3.2 Assignment Model

When a student opens a tests.arr file in Examplar_{\alpha}, the IDE must fetch the appropriate wheat and chaff implementations. Examplar_{\alpha} achieves this by extending CPO with a limited notion of assignment. An assignment, in the context

\textsuperscript{5} \url{https://github.com/brownplt/pyret-lang/wiki/Error-Reporting,-July-2016}
of Examplar, is merely a world-readable Google Drive folder created by an instructor that provides wheats, chaffs, and starter files in a prescribed layout. At the top-level of this folder, there must exist:

1. A folder named “wheat”, containing wheat implementations.
3. A file ended with the substring “-tests.arr”, that serves as an initial template file for students’ test suites.

Figure 7 depicts the assignment folder of a contrived assignment, “fib”, that asks students to write examples for a function that produces elements from the Fibonacci sequence.

The wheat and chaff folders are each filled with implementations, represented either as Pyret source code (.arr) or compiled Pyret modules (.js). Course staff typically used the source-code form for assignment development, then replaced those files with compiled Pyret modules before releasing the assignment. Distributing compiled Pyret modules rather than source code provides both an end-user performance improvement, and prevents students from merely copying the source code of the assignment’s implementations.
OTHER WAYS TO INCENTIVIZE EARLY TESTING

Instructors have explored a variety of interventions to incentivize and train students to test early and effectively. These approaches vary considerably in what students do and how feedback is provided.

4.1 PEER REVIEW

In peer-review based interventions, students improve the quality of their test suites via interactions with each other.

4.1.1 Code Defenders

Code Defenders is a multiplayer game that pits students against each other in opposing teams: attackers, who iteratively introduce bugs into a program, and defenders, who attempt to detect each successive round of bugs by writing test cases. Examplar is similar to Code Defenders, except students exclusively take the role of defenders, and course staff (who prepare correct and buggy implementations in advance of putting out the assignment) serve as attackers. Whereas Code Defenders is only usable in a multi-player setting, students may use Examplar independently, without concurrent participation from anyone else. Code Defenders’s use in classes has not yet been rigorously evaluated.

4.1.2 Captain Teach

Captain Teach is a non-adversarial interface for students to peer-review their tests and code. The peer review is in-flow, because it is done while the assignment is in-progress. Unlike Code Defenders, the processes of peer review and development in Captain Teach are not iterative. Later work emphasized the creation

---

1 This chapter adapts content previously published by the author in Executable Examples for Programming, Problem Comprehension, Will Students Write Tests Early Without Coercion?, and Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors.
of a small number test cases ("essential examples") for peer review and early submission. As with our work, these sweeps were graded programmatically in a wheat–chaff assessment style. Unlike our work, this feedback was not available to students on-demand.

4.2 TESTING QUIZES

In testing quiz interventions, students must complete gaps in instructor-authored, incomplete test cases.

4.2.1 Prather et al.

Prather et al. [23] asked students, before they began their implementation, to correctly predict the output of the specified function for a given input. As with Examplar, this intervention provides an opportunity for students to verify that their understanding of the problem matches the prompt. However, Examplar differs from Prather et al.’s work in several key ways. First, Examplar requires that students develop their own input data for examples. Second, in addition to being valid, Examplar-assessed examples must also be thorough explorations of the problem’s interesting facets. Third, our students were welcome to use Examplar at any point in their development process (or not at all); Prather et al.’s intervention was strictly situated between reading the problem prompt and developing a solution.

4.3 EXTERNAL ON-DEMAND AUTOMATED FEEDBACK

4.3.1 Ante

Michael Bradshaw’s Ante assessment framework [27] adopts the classifier perspective in all-but-name:

A test is a way of determining if a particular implementation meets all of the specifications. So a perfect test will only accept implementations that are correct and reject all other implementations.

In the Ante test assessment model, the instructor authors N implementations, 1 correct one, and N − 1 buggy ones. A student’s test is deemed “correct” if it correctly classifies all N implementations. In contrast, we further decompose “correctness” into validity and thoroughness.

Unlike Examplar, Ante additionally provides students with feedback on their implementation quality. This is assessed by subjecting the student’s implementa-
tion to an instructor-authored test suite. In the course-contexts where Examplar has been used, this same, test-suite-based mechanism is used to provide students’ with a final grade on their implementation.

Ante withholds this implementation-quality feedback until the student submits a test suite with perfect correctness. This restriction is intended to encourage students to focus on testing before implementation. Similarly, Examplar withholds thoroughness feedback if invalid tests are detected. However, Examplar does not require students to catch all chaffs before getting implementation feedback (i.e., the feedback students get from running their own test cases against their implementation).

4.3.2 Web-CAT

In 2003, Web-CAT, an online automated assessment system, computed three measures relating to student testing [28]:

1. *test validity*, how consistent the student’s tests are with the problem;

2. *test completeness*, how thoroughly the student’s tests cover the problem; and

3. *code correctness*, how correct the student’s code is.

Web-CAT assesses validity by counting the proportion of the student’s tests which accept an instructor-authored reference implementation; assesses completeness by computing the coverage of the student’s tests on their own implementation; and assesses code correctness by computing the proportion of the student’s tests which pass their own implementation.

We share Web-CAT’s nomenclature and approach of assessing validity, but differ in our treatment of thoroughness: we compute the sensitivity of test suites on sets of known-buggy implementations; Web-CAT computes the suite’s coverage on a particular implementation. Accordingly, we adopt the term “thoroughness” rather than “completeness”, because while it is possible for a test suite to have complete coverage of a particular implementation, it is impossible for any test suite to completely catch all possible buggy implementations.

To ensure that “no aspect of the approach can be ignored”, Web-CAT multiplies the measures together to produce a single, final grade. Similarly, we do not compute the thoroughness of invalid tests; Examplar will refuse to compute thoroughness if any invalid tests are present. However, Web-CAT’s mechanism, as described, appears to have a vulnerability: a student can easily cover the reference implementation in their tests with no risks of invalidity by simply making many method calls with varied inputs, but accepting any output.
Unlike Web-CAT, Examplar does not provide code correctness feedback. In the course contexts where Examplar has been used, such feedback is provided to students during final grading, albeit with a different mechanism: students’ implementations are assessed with instructor-authored tests. This difference comes with both advantages and disadvantages. Our use of instructor-authored tests to assess student code ensures that even students who test poorly still receive a thorough assessment of their code. Web-CAT’s use of the student’s own tests for this purpose provides students with additional incentive to write through tests; i.e., to receive thorough feedback on their implementation.

4.4 EXTERNAL ON-DEMAND AUTOMATED FEEDBACK

4.4.1 Learn-OCaml

Hameer & Pientka [47] present a substantial extension of the Learn OCaml IDE to provide on-demand feedback of various kinds, including wheat–chaff style evaluation of test suites. This work could be the basis for an exploration of executable examples in OCaml. In its current form it does not fill this role due to several key differences with our work:

1. Hameer & Pientka’s environment presents students with a full-fledged grade report; we do not provide students with on-demand feedback of their grade.

2. Hameer & Pientka’s environment provides feedback on students’ implementations; our students must evaluate their implementations with their own test cases.

3. Hameer & Pientka’s environment does not distinguish between tests of the spec and tests of implementations-specific behavior; our method does.

4. Hameer & Pientka’s environment has not yet been evaluated with research (i.e., they do not attempt to evaluate the impact of the IDE on students quantitatively or qualitatively).
PEDAGOGIC CONTEXT

We primarily assess Examplar in the context of Brown University’s CSCI0190: An Accelerated Introduction to Computer Science (csci0190): chapter 6 contrasts final submissions made by students in the Fall 2017 and Fall 2018 offerings of csci0190; chapter 8 analyzes the programming process of students in the Fall 2019 offering of csci0190; chapter 8 considers the help-seeking behavior of students in the Fall 2020 offering of csci0190. To avoid needless repetition in these chapters, this chapter describes factors common across the considered offerings.

5.1 COURSE STRUCTURE

The primary course activity of csci0190 was programming projects. For all programming projects, students were given a specification in prose and required to submit an implementation consistent with that specification. For most projects, students additionally submitted a test suite. The ordering (table 1), length (table 2) and substance (section 5.4) of assignments varies slightly between years — these differences are noted in subsequent chapters when appropriate.

5.2 ASSIGNMENT STRUCTURE

For all programming projects (except where otherwise noted), students were required to submit both a code file (containing their implementation of the problem specification and white-box tests) and a tests file (containing black-box tests consistent with the problem specification). In Fall 2019 and 2020, students additionally submitted a common file—a shared dependency of both the code and tests files—containing helper functions and testing data used in both code and tests. Prior to the introduction of common, students were advised to simply duplicate common functions in both their code tests file.

1 This chapter adapts content previously published by the author in Executable Examples for Programming Problem Comprehension [1], Will Students Write Tests Early Without Coercion? [2], and Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors [3].
Table 1: The ordinal of assignments between offerings of csci0190. An assignment is listed as having multiple ordinals if it was broken up into sub-assignments prior to the Fall 2020 offering of csci0190.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017</th>
<th>2018</th>
<th>2019</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocDiff</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Nile</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sortable</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>DataScripting</td>
<td>4, 6</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Oracle</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Filesystem</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Updater</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>ContFracs</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>TweeSearch</td>
<td>na</td>
<td>na</td>
<td>11, 12, 16</td>
<td>9</td>
</tr>
<tr>
<td>JoinLists</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>TourGuide</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>MST</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>MapReduce</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>FluidImages</td>
<td>13, 14</td>
<td>13</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2: The length of assignments between offerings of csci0190, in days. The lengths of projects divided into multiple assignments are summed together.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017</th>
<th>2018</th>
<th>2019</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocDiff</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Nile</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Sortable</td>
<td>3 + 3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DataScripting</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Oracle</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Filesystem</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Updater</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>ContFracs</td>
<td>na</td>
<td>na</td>
<td>2 + 3 + 3</td>
<td>4</td>
</tr>
<tr>
<td>TweeSearch</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>JoinLists</td>
<td>14</td>
<td>14</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>TourGuide</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>MST</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>MapReduce</td>
<td>7 + 4</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>FluidImages</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
5.3 Grading

csci0190’s grading has both automated and manual components. Manual grading does some correctness checking, but focuses much more on design and stylistic issues (and also evaluates non-code aspects like Big-O analyses).

Each of the code and test files are separately auto-graded. The code is graded against a staff-authored test suite. This feedback is withheld from students until after the assignment deadline.

Students tests are graded against wheats and chaffs (as described in chapter 2). Students can run Examplar at any time before or after the deadline; these are not monitored or graded. Because students can spend too long catching chaffs by writing ever-more-complex tests we included only a small number (4–6) of chaffs in Examplar. However, they are expected to write much more thorough test suites to identify bugs in their implementations. Therefore, during grading, their test suites are run against many more chaffs (upwards of 20). Their testing grade is based on how well their valid tests do against these chaffs.

In all offerings of the course, feedback on the students’ implementation is withheld until after the assignment due date. In the Fall 2017-2019 offerings of csci0190, wheat and chaff feedback (except for that provided by Examplar) was withheld from students until after the assignment due date. In the Fall 2020 offering of csci0190, students could receive the full set of wheat and chaff feedback (with wheat and chaff names redacted) upon homework submission (with unlimited resubmits allowed); unredacted feedback was withheld until after the assignment due date.

5.4 Assignments

The assignments of Fall 2020, most of which are archetypal of the preceding years, are as follows:

5.4.1 DocDiff

Students implemented and tested a function computing a case-insensitive document similarity metric using a bag-of-words model [48]:

```haskell
fun overlap(doc1 :: List(String), doc2 :: List(String)) → Number:
  ...
end
```
The domain of overlap is additionally constrained in the assignment specification to non-empty documents (i.e., non-empty lists). The result of this procedure on inputs where either doc1 or doc2 are empty is unspecified.

Accordingly, we constructed two wheats with which we assessed the validity of overlap tests: one producing 0 on empty inputs, and another producing an error.

5.4.2 Nile

Students implemented and tested a rudimentary book recommendation system. At the heart of this assignment was a File datatype, defined in support code:

```
data File:
  | file(name :: String, content :: List(String))
end
```

The content field of files is defined as a list of book names. If two books in two different files have the same name, they conceptually represent the same book. The content field contains no duplicate elements.

Next, students implement a function that, given a book, recommends another book:

```
data Recommendation(A):
  | recommendation(count :: Number, content :: List(A))
end

fun recommend(title :: String, book-records :: List(File)) → Recommendation(String)
  ...
end
```

As with File, the content field of a Recommendation is a list of book titles. The order of book titles in this list is unspecified behavior. Two wheats were provided, providing these recommendations in opposite orders.

Finally, students implement a function that recommends popular pairs of books:

```
data BookPair:
  | pair(book1 :: String, book2 :: string)
end

fun popular-pairs(records :: List(File)) → Recommendation(BookPair):
  ...
end
```

How pairs are ordered (i.e., which book is book1 and which is book2) is unspecified. To simplify testing, the assignment support code overrides equality for BookPairs to ignore ordering within pairs.
5.4.3 Sortacle

Students implemented a testing oracle of functions that purport to sort lists of people in ascending order by age. This assignment fails a prerequisite of our assessment model: the correctness of a testing oracle is not a binary property (see section 2.2.1). It is generally impossible for a testing oracle to perfectly identify buggy functions-under-test; even the most thorough oracles will still mislabel subtly buggy procedures-under-test as correct.

5.4.4 DataScripting

From 2018–2020, DataScripting was a multi-part assignment consisting of seven, self-contained problems for which students submitted implementations, but not tests:

1. Palindrome Detection Modulo Spaces and Capitalization
   
   Implement the predicate \texttt{is-palindrome :: String} \rightarrow boolean, which is satisfied only if the given string (notwithstanding spaces and case) is a palindrome.
   
   This specification does not admit any unspecified behavior.

2. Sum Over Table
   
   «Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths. Design a program \texttt{sum-largest} that consumes a table of numbers and produces the sum of the largest item from each row. Assume that no row is empty.»
   
   This specification admits unspecified behavior in the presence of empty rows.

3. Adding Machine
   
   «Design a program called \texttt{adding-machine} that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.»
   
   This specification does not admit any unspecified behavior.

4. The BMI Sorter
   
   «A personal health record (PHR) contains four pieces of information on a patient: their name, height (in meters), weight (in kilograms), and last recorded heart rate (as beats-per-minute). A doctor’s office maintains a list of the personal health records of all its patients. [...] Design a function called \texttt{bmi-report} that consumes a list of personal health records (defined above) and produces a report containing..."
a list of names (not the entire records) of patients in each BMI classification category. The names can be in any order.»

5. Data Smoothing

«In data analysis, smoothing a data set means approximating it to capture important patterns in the data while eliding noise or other fine-scale structures and phenomena. One simple smoothing technique is to replace each (internal) element of a sequence of values with the average of that element and its predecessor and successor. [...] Design a function data-smooth that consumes a list of PHRs and produces a list of the smoothed heart-rate values (not the entire records).»

This specification does not admit any unspecified behavior.

6. Most Frequent Words

«Given a list of strings, design a function frequent-words that produces a list containing the three strings that occur most frequently in the input list. The output list should contain the most frequent word first, followed by the second most frequent, then the third most frequent. If two words have the same frequency, put the shorter one (in character length) first. You may assume that:

a) the input will have at least three different words
b) all characters are lowercase letters (there will be no numbers, punctuations, or white spaces)
c) multiple words with the same frequency will have different lengths

»

This specification admits unspecified behavior if the aforementioned assumptions are violated, but the Examplar instance for this assignment did not include multiple wheats, and thus was unable to detect tests of unspecified behavior.

7. Earthquake Monitoring

Students implement a function that parses a list of numbers into dates and associated data points, and summarizes that data by date. Dates are eight digit numbers (e.g., “20151004”); data points are numbers ranging from 0 up-to 500. Dates are followed by one-or-more data points, occur within the same year, and are ordered chronologically.

This specification admits unspecified behavior if the aforementioned assumptions are violated, but the Examplar instance for this assignment did not include multiple wheats, and thus was unable to detect tests of unspecified behavior.
For the 2018 offering of the course, wheat–chaff feedback was available in Examplar for all sub-problems except BMI Sorter. The 2019 and 2020 offerings of the course did not offer any wheat–chaff feedback due to a combination of technical and coordination issues.

5.4.5 Oracle

Students implemented a testing oracle for functions solving the stable marriage problem of matching companies to candidates.

This assignment challenges students to apply pbt to a problem they most probably cannot solve. At the time Oracle is assigned, the course has not introduced the necessary algorithmic techniques, and we expressly do not expect students to solve it. The assignment provides students with the source code of a correct implementation, which can use refer to in addition to using it to test their oracle.

As with Sortacle, on-demand wheat–chaff feedback was not available to students on this assignment.

5.4.6 Filesystem

Students implemented and tested a variety of rudimentary Unix-style commands for traversing a (in-memory) file structure with mutually-dependent datatypes [49]. These procedures consumed a representation of a filesystem, defined by a combination of two types. First, each File has a name and stores some content:

```plaintext
data File:
  | file(name :: String, content :: String)
end
```

Second, each directory has a name and contains additional sub-directories and files:

```plaintext
data Dir:
  | dir(name :: String, ds :: List(Dir), fs :: List(File))
end
```

The domain of dir has an additional constraint: the files in fs must have distinct names. The behavior of the filesystem traversal functions is unspecified on malformed filesystems. To detect tests of invalid inputs, the wheats for this assignment differ in their behavior on malformed inputs.

One of the filesystem functions students implement is the procedure fynd, which consumes a filesystem and a filename, and produces a list of paths to all files in the filesystem matching that name:
type Path = List(String)

fun fynd(a-dir :: Dir, fname :: String) → List(Path):
  ...
end

The order of the result returned by fynd is unspecified. To detect over-constrained tests of fynd’s output, the wheats for this function produce outputs in opposite orders.

5.4.7 Updater

Students derived, implemented, and tested Huet Zippers [50], a data structure encoding an n-ary tree which can be efficiently traversed and updated with a virtual cursor.

While the traversal functions specified by this assignment do not possess unspecified behavior, the Cursor datatype itself is entirely up to students to design and implement. As such, tests that make assumptions about the structure of Cursor are invalid. The wheat overrides the equality method of Cursors to always return false.

5.4.8 Continued Fractions

Students implemented and tested a stream-based representation of continued fractions and a number of functions operating over that representation.

The take routine,

fun take(T:s :: Stream(T), n :: Number) → List(T):
  ...
end

given a Stream, extracts a prefix of the specified size, n, as a List. The behavior of this function on negative values of n is unspecified. One wheat produces an error on such inputs; the other produces an empty list.

The repeating-stream routine,

fun repeating-stream(numbers :: List(Number)) → Stream(Number):
  ...
end

is given a list of numbers and produces a stream that repeats that list indefinitely. The behavior of this function on empty input lists is unspecified. One wheat produces an error on such inputs; the other loops infinitely. A variant of this function (repeating-stream-opt) that produces a stream of Option(Number) elements, shares the same unspecified behavior.
The function-stream routine (and its variant function-stream-opt),

```plaintext
fun function-stream(coefficients :: Stream(Number)) → Stream(Number):
    ...
end
```

both consume a stream of continued fraction coefficients, and produces a stream of increasingly-accurate approximations. The given coefficients must be non-negative integers; the behavior of this function on coefficients not meeting this condition is unspecified. Tests involving invalid coefficients were not fully detected by the wheats for two reasons. First, because function-stream processes its (potentially infinitely many) given coefficients lazily; it is not until its produced stream is consumed that invalidity can be detected. Second, even when consumed, both wheats raised an error on certain invalid inputs (as opposed to one wheat producing an error, and the other producing a value); consequently, this invalid test was marked as valid by Examplar:

```plaintext
check:
    take(function-stream(repeating-stream([list: 1, -2])), 10) raises ""
end
```

### 5.4.9 TweeSearch

In this three-part assignment, students successively implement three variations of a fuzzy search routine for Tweet. The implementation details of the Tweet datatype vary between sub-parts, but the signature of the search routine is consistent:

```plaintext
fun search(
    needle :: Tweet,
    haystack :: List(Tweet),
    threshold :: Number
) → List(Tweet):
    ...
end
```

Given a “needle” Tweet to search for, a “haystack” to search within, and a given threshold of relevance to the needle that prospective matches must surpass, the search function produces a list of matching Tweets in descending order of relevance to the haystack.

In all three sub-parts of this assignment, Tweets have content, represented as a String. This string must be non-empty. The threshold parameter of search must range between zero and one (inclusive). To detect tests that violate these constraints, one wheat produces an error for unspecified inputs; the other produces a value.

The query results are specified by the assignment to be arranged in descending order of relevance. The relative ordering of equally-relevant tweets is unspecified.
To detect over-constrained tests that assert a particular ordering of equally-relevant results, the wheats produce equally-relevant sequences of tweets in opposite orders.

### 5.4.10 JoinLists

Students implement several routines operating over a *conc-tree list* data structure (called a “JoinList” by this assignment):

```haskell
data JoinList(T):
    | empty-join-list
    | one(elt :: T)
    | join-list(list1 :: JoinList(T),
              list2 :: JoinList(T),
              length :: Number)
end
```

A join-list is invalid if its `length` field is not equal to the sum of the number of elements stored by `list1` and `list2`. To detect tests using invalid `JoinList`s as inputs, the wheats differ on whether they produce an error or a value when given invalid inputs.

One of the routines implemented by students is `j-nth` which produces the n\textsuperscript{th} element of the list. The given n must range between 0 (inclusive) and the length of the given list (exclusive). The behavior of `j-nth` on out-of-range inputs is unspecified. To detect tests providing invalid ns, one wheat produces an error on such inputs and the other produces a value.

Students also implement `j-max`, which produces the maximum value in a non-empty list:

```haskell
fun j-max(A)(
    jl :: JoinList(A)%is-non-empty-jl,
    cmp :: (A, A → Boolean)
) → A:
    ...
end
```

Which value is produced if the given `JoinList` contains equally maximal values is unspecified. On such lists, the wheats differ in which maximal value is produced.

Students also implement `j-sort` which sorts the given `JoinList` using a given comparator:

```haskell
fun j-sort(A)(
    cmp-fun :: (A, A → Boolean),
    jl :: JoinList(A)
) → JoinList(A)
    ...
end
```
The relative ordering of “equal” values (per cmp-fun) is unspecified; the wheats sort sequences of equal values in opposite orders.

An additional, unchecked restriction constrains the domain of j-max and j-sort: the given comparator function must define a valid total ordering of elements. Unfortunately, it is impossible to programmatically detect whether an arbitrary comparator is valid in this regard. Wheats cannot be used to detect tests that supply invalid comparators.

5.4.11 TourGuide

Students implement two routines over a Graph of named Places. First, dijkstra, which consumes the Name of a starting point in the given Graph, and produces a set of shortest paths to every other place in the Graph:

```haskell
fun dijkstra(start :: Name, graph :: Graph) → Set(Path):
    ...
end
```

The result when two equally-shortest paths exist to a place is unspecified. One wheat selects the lexicographically first candidate; the other selects the lexicographically last candidate.

Second, campus-tour, is a routine that consolidates a set of Tours (where each Tour is a name plus a set of Places), a start position, and an Graph representing the underlying topology those tours occur within, and produces a Path that visits all of the places in the input set of tours:

```haskell
fun campus-tour(
    tours :: Set(Tour),
    start-position :: Point,
    campus-data :: Graph) → Path:
    ...
end
```

The assignment specifies that this routine should select destinations greedily; i.e., choose the next place to visit by moving toward the closest, unvisited place in the provided set. Again, the output when two shortest paths exist is unspecified—one wheat selects the lexicographically first candidate; the other selects the lexicographically last candidate.

The behavior of this function when given tours with duplicate names is unspecified, as is its behavior when the tours include stops that are not within the campus-data Graph—one wheat produces values for such inputs; the other raises errors.
5.4.12 MST

Students implemented both Prim’s ([mst-prim]) and Kruskal’s ([mst-kruskal]) minimum spanning tree algorithms, as well as a “sort-of oracle”. This “sort-of oracle” consists of:

- **generate-input**, which consumes a number and produces a **Graph** of that size.
- **mst-cmp**, which consumes a **Graph**, and two (possibly identical) purported minimum spanning trees within that graph, and produces **true** if both purported minimum spanning trees exist in the given graph and have the same total weight.
- **sort-o-cle**, which consumes two purported implementations of minimum spanning tree, and produces **true** if they produce equally-minimal spanning trees on a number of randomly generated graphs.

The behavior of **mst-cmp**, **mst-prim** and **mst-kruskal** on unconnected input graphs is unspecified. The output of **mst-prim** and **mst-kruskal** on graphs that admit multiple minimum spanning trees is unspecified. The order of vertices in the produced minimum spanning trees are unspecified. The behavior of **generate-input** on negative inputs is unspecified.

5.4.13 MapReduce

Pairs of students implemented and tested the essence of MapReduce ([51]) (implemented sequentially), and applied it to multiple problems. This included re-doing some previous assignments (including Nile) in terms of the MapReduce paradigm, using their implementation. The Nile-reprise subproblem of this assignment retained the same domain constraints as Nile, and these invalid tests of out-of-domain inputs were detected with multiple wheats in much the same way. The assignment specification notes that the output orders of the map-reduce related functions (namely anagram-reduce, anagram-map, and map-reduce) are unspecified, and provides students with two utility functions ([lst-same-els] and [recommend-equiv]) to assist them with testing. Over-constrained student tests of output order are detected with two wheats, each of which produces outputs in opposite orders from the other.

5.4.14 Fluid Images

Students implement image seam carving ([52]) twice: with memoization (**liquify-memoization**) and with dynamic programming (**liquify-dynamic-programming**). Each of these func-
tions consume an `Image` and a `Number` of seams to carve away, and produces a carved image. This input `Number` must range between zero and the width of the input `Image` (in pixels); the behavior of these procedures on inputs not meeting these conditions is unspecified. One wheat produces an error if these conditions are violated; the other produces the input image.
Are Executable Examples Appealing, Helpful?¹

In the presence of Examplarα interface, we ask: *Did students in an accelerated introductory course (csci0190)*...

**rq 1**: ...choose to use Examplarα?

**rq 2**: ...ultimately submit more or better test cases?

**rq 3**: ...ultimately submit more correct implementations?

6.1 Method

We deployed Examplarα in fall 2018 in an accelerated introduction to computer science course offered at Brown University.

6.1.1 Pedagogic Context

The course instructs students on the design recipe, algorithm and data structure design, and algorithm (big-O) analysis. The course, its assignments and the availability of executable example feedback are described more fully in chapter 5.

Course Structure The 2018 offering of the course featured fourteen programming projects. For all of these projects, students were given a prose specification and were required to submit an implementation consistent with that specification. For twelve of these projects, students additionally submitted a test suite. We provided Examplarα on the ten projects that met the expectations outlined in section 2.2.1. The projects included constructing a recommendation engine, modeling a filesystem [49], deriving Huet zippers [50], and seam carving [52].

¹ This chapter adapts content previously published by the author in *Executable Examples for Programming Problem Comprehension* [11].
Demographics  Sixty-seven students completed the course. Most were first-year students, approximately 18 years old, with some prior programming experience (though not typically with prior testing experience). About $\frac{1}{6}$ identified as female. Admittance to the course required successful completion of four assignments roughly corresponding to the first fifth of HTDP.

Pedagogy  The instructor asked students to follow the entirety of the design recipe while completing all programming projects. However, in requiring the submission of only a final implementation and a test suite, the course essentially enforced only the last two steps of the design recipe.

Previous iterations of the course attempted to apply the idea of a “sweep” [53]: graded examples due several days before the final submission deadline. The fast pacing of programming projects precluded this requirement for most assignments, but it was hoped the habit of early example-writing would stick. However, from the guilty admissions of former students,² we believed that for projects lacking this early deadline, students authored most (if not all) of their assertions after developing their implementation. We hoped Examplarα would be an effective alternative to strict early deadlines.

6.1.2  RQ1: Do students use Examplarα?

To determine whether students used Examplarα, we monitored their use of the tool, instrumenting Examplarα to log the username and suite contents each time a user clicked Run Tests.

RQ1.1: ...When it is not required?  We wanted to see the degree to which students used Examplarα on their own volition; we thus did not force students to use the tool. However, we feared students might not try the tool at all merely as a matter of lack of exposure. We therefore required that students use Examplarα for the first assignment, but made usage optional thereafter. To judge whether students valued Examplarα on their own volition, we compare submission volume for this first assignment to that of subsequent assignments.

RQ1.2: ...When no final test suite is required?  On one assignment, DataScripting, students were not required to submit a test suite, but Examplarα was still provided. This assignment was a collection of seven, small, independent programming problems (adapted from Fisler et al. [18]). Students submitted independent implementation files for each part, and we provided independent Examplarα instances for six of the seven parts. To judge whether students

² In particular, the former students who were hired to be 2018’s TA staff!
Table 3: The position and duration of the comparable assignments in each year.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017 Ordinal</th>
<th>2017 Days</th>
<th>2018 Ordinal</th>
<th>2018 Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocDiff</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Nile</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>DataScripting</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Filesystem</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Updater</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>JoinLists</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>MapReduce</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

used Examplar$_x$ when no final test suite was required, we compare volume of submissions for this assignment to the other assignments.

**rq 1.3: throughout their development process?** The Examplar$_x$ usage logs provide only a partial view of students’ overall development process; students still needed to use Pyret’s usual editing environment to develop their implementations and to run their test suite against their own implementations. Instrumenting Pyret’s usual editor was not feasible. The usage logs therefore do not tell us how students used Examplar$_x$ in relation to their other development progress. We supplement our understanding with a voluntary survey prompting students for feedback to “help us evaluate if and how we use Examplar$_x$ in future semesters”. In this survey, we asked students to self-report their use of Examplar$_x$, relative to their progress in developing their implementations.

6.1.3 Do final submissions change?

To determine whether Examplar$_x$ induced changes in students’ final submissions, we looked at the 2017 offering of the course as a point of comparison. Aside from the introduction of Examplar$_x$, the 2018 offering of the course was virtually unchanged from the 2017 offering. Of the fourteen programming projects in the 2018 offering, thirteen appeared in the 2017 offering. Both offerings used the same entry process, featured similar lectures (which were held at the same times), and provided similar resources for students. The student demographic was almost nearly identical (except with 76 students, resulting in more total implementations and tests).

Naturally, there were some changes; we did not restrict the 2018 course staff from correcting significant issues as they saw fit. Nevertheless, of the ten assignments for which Examplar$_x$ was provided, seven were functionally identical to their 2017 offering (table 3); i.e., we are able to meaningfully assess the submissions
for both years using identical wheats, chaffs, and test suites. We use subsets of these comparable assignments to judge whether the quality of students’ final test suites and implementations improved. In section 6.3 we discuss the limitations of this evaluation approach and why we did not perform a more tightly controlled study.

rq 2: Do test suites change?

Of the comparable assignments, five required the submission of a test suite. We use these five assignments to judge whether final test suite quality improved. This subset of assignments is comprised of 320 final test suite submissions for 2017, and 269 final test suite submissions for 2018. We assess the quality of these submissions using identical wheats and chaffs. We consider the size, validity and thoroughness of test suites on these assignments in turn:

rq 2.1: Does test suite size increase? To determine whether final test suite size increased, we contrast the number of tests in suites from each year. We hypothesized that, by gamifying the testing experience, Examplarα would induce students to write more tests. We perform a two-sample t-test to determine if the average number of test cases significantly differs between years.

rq 2.2: Does validity improve? We hypothesized that, in aggregate, the validity of final test suites would improve significantly from 2017 to 2018. Examplarα’s feedback on validity is complete; i.e., if a test suite accepts all of the wheats in Examplarα, it will accept all of the wheats in the autograder used for final submissions. We sort the implementations for each year into the dichotomous categories of valid and invalid, and perform a χ² test to determine if the proportion of valid test suites differ significantly.

rq 2.3: Does thoroughness decline? The chaffs used to assess final test suites included both mistakes of logic and implementation errors. However, Examplarα only included chaffs targeting the former, so it is conceivable that students could misinterpret catching all chaffs within Examplarα as having “finished” their test suite. We therefore must check whether the thoroughness of students’ test suites declined. We compute the thoroughness of each final test suite (section 2.1) and, conditioned on observing a decrease in the proportion of chaffs caught between years, perform a χ² test to determine if the difference is significant.
rq 3: Do implementations change?

We hypothesized that the direct aid provided by Examplar\textsubscript{α} for test development would indirectly benefit students’ implementations. All seven of the comparable assignments required the submission of implementations. This set of assignments is comprised of 622 implementations from 2017, and 522 from 2018. We sort the implementations for each year into the dichotomous categories of correct and buggy using an instructor-authored test suite, and perform a $\chi^2$ test to determine if the proportion of correct implementations significantly differ.

6.2 Results

We present our findings for each of the research questions stated in section 6.1.

6.2.1 rq 1: Did students use Examplar\textsubscript{α}?

Students used Examplar\textsubscript{α} extensively on all assignments, clicking Run Tests a total of 26,211 times. Figure 8 illustrates the distribution of Examplar\textsubscript{α} submissions per-student for each of the assignments where Examplar\textsubscript{α} was provided.

rq 1.1: ...when it was not required? Yes. Students used Examplar\textsubscript{α} extensively even after the requirement to use it was dropped. The median Examplar\textsubscript{α}-submissions-per-student for DocDiff of 22 (the first and only assignment for which Examplar\textsubscript{α} use was required) was less than that of any other assignment.\textsuperscript{3} Only a small number of students elected to not use Examplar\textsubscript{α} thereafter: 4 students on DataScripting, 3 on FileSystem, and 1 on Updater, MapReduce, TourGuide, and FluidImages.

rq 1.2: ...when no final test suite was required? Yes. Figure 9 illustrates the distribution of the number of Examplar\textsubscript{α} submissions per-student for each of DataScripting’s parts. Of 67 students who submitted an implementation for at least one part, 64 used it for at least one part and 48 used it for every part for which they submitted an implementation. Examplar\textsubscript{α} usage for this assignment is particularly notable as students were given only two days to complete its seven parts. Interpreted as a whole, Examplar\textsubscript{α} usage for this assignment was on par with that for the other assignments; the median student submitted 33 suites to Examplar\textsubscript{α} for DataScripting.

\textsuperscript{3} The individual parts of DataScripting received fewer submissions-per-student than DocDiff, but each was a significantly smaller problem than any other in the course.
Figure 8: For each assignment: a combined violin and box-and-whiskers plot illustrating the volume of Examplar $\alpha$-submissions-per-student for each assignment. The center line in each box represents the number of times the median student clicked Run Tests. The whiskers extend to the most extreme data lying within 1.5 times the interquartile range. Points indicate the exact number of submissions of outlying students.
Figure 9: Example submissions per-student for each of DATASCRIPTING parts, rendered in the same manner as fig. 8.
Table 4: Did you use Examplarα to write examples or tests Before, During, and After completing your implementation? (Higher percentages are shaded darker.)

<table>
<thead>
<tr>
<th>Usage</th>
<th>Before</th>
<th>During</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rarely, or not at all</td>
<td>4.3%</td>
<td>4.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>A few times</td>
<td>21.7%</td>
<td>13.0%</td>
<td>13.0%</td>
</tr>
<tr>
<td>About half of the time</td>
<td>39.1%</td>
<td>17.4%</td>
<td>17.4%</td>
</tr>
<tr>
<td>Most of the time</td>
<td>21.7%</td>
<td>26.1%</td>
<td>17.4%</td>
</tr>
<tr>
<td>Almost always, or always</td>
<td>13.0%</td>
<td>39.1%</td>
<td>52.2%</td>
</tr>
<tr>
<td>Unsure</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

**rq 1.3:**...throughout their development process? Probably. Twenty-three students (approximately a third of the students enrolled in the course) provided feedback on their Examplarα usage in the voluntary course feedback survey. When asked, “Did you use Examplarα [before, during, after] developing your implementation?”, a majority of students indicated they used Examplarα at least “about half the time” at all stages. Self-reported Examplarα usage (table 4) increased as implementation development progressed. Of course, students’ self appraisal of their own testing diligence should be regarded with some skepticism, especially on a non-anonymous survey distributed a month after the course ended.

6.2.2 Did final submissions change?

Yes. In aggregate, the quality of both test suites and implementations improved from 2017 to 2018.

**rq 2:** Did test suites change?

Yes. In aggregate, the validity of final test suites significantly improved, without any degradation in their thoroughness, on the five comparable assignments. Curiously, there was no significant difference in the size of students’ test suites. We consider the size and quality of these suites in turn:

**rq 2.1:** Did size increase? No, the number of assertions in final test suites was approximately equal (fig. 10). With a Welch’s t-test, we determined that the difference in average size between suites in 2017 and 2018 was not significant (t(523.79) = 0.66871, p = 0.504).

**rq 2.2:** Did validity improve? Yes. In aggregate, final test suites in 2017 were 4.8 times more likely to be invalid than suites in 2018. Table 5 illustrates the
proportion of invalid test suites for each of the comparable assignments. Our $\chi^2$ test with Yates’ continuity correction revealed that the validity of test suites on comparable assignments significantly differed by year ($\chi^2(1, N = 589) = 52.373$, $p < 0.01$, $\phi = 0.303$, the odds ratio is 0.16).

Why were fewer final test suites invalid in 2018? There are three important causes of invalidity:

1. If a suite accepts some—but not all—wheats, it is almost certainly asserting underspecified behavior. Final test suites in 2017 were 12.9 times more likely to have this form of invalidity than suites in 2018.

2. If a suite fails all of the wheats because one or more assertions rejected the wheats, it may be that the student tested either underspecified behavior or incorrect behavior. Final test suites in 2017 were 6.1 times more likely to have this form of invalidity than suites in 2018.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Invalid</td>
<td>n</td>
</tr>
<tr>
<td>DocDiff</td>
<td>29.7%</td>
<td>91</td>
</tr>
<tr>
<td>FileSystem</td>
<td>30.3%</td>
<td>76</td>
</tr>
<tr>
<td>Updater</td>
<td>20.0%</td>
<td>75</td>
</tr>
<tr>
<td>JoinLists</td>
<td>17.9%</td>
<td>39</td>
</tr>
<tr>
<td>MapReduce</td>
<td>64.1%</td>
<td>39</td>
</tr>
<tr>
<td>Aggregate</td>
<td>30.3%</td>
<td>320</td>
</tr>
</tbody>
</table>
3. A suite may fail to compile or run any of its assertions. This is often indicative of the student failing to follow the template for test suite submission (e.g., they tweaked the imports). Final test suites in 2017 were 1.3 times more likely to have this form of invalidity than suites in 2018.

Table 6 details the number of final test suites of each form of invalidity for the comparable assignments in 2017 and 2018.

rq 2.3: Did thoroughness decline? No. Test suites in 2018 were no less thorough (table 7). We can therefore be confident that the aforementioned gains in validity did not come at the expense of thoroughness. As we do not observe any decrease in thoroughness, we do not perform a $\chi^2$ test.

rq 3: Did implementation quality improve?

Inconclusive. Our $\chi^2$-square test with Yates’ continuity correction revealed that the overall proportion of correct implementations (table 8) did not strongly significantly differ by year ($\chi^2(1, N = 1144) = 2.94, p = 0.086, \phi = 0.053$, the odds ratio is 0.8).

6.3 Limitations

Threats to Internal Validity We feel it is reasonable to attribute the differences we observed between years to Examplar2 because of the extensive similarities between the offerings. However, it may be that these different cohorts of students behaved differently due to an external factor. Ideally, we would convince ourselves that this is unlikely by considering submissions from additional years. This has practical difficulties. First, course changes naturally accumulate; few of the assignments in 2016 are functionally identical to those of 2018. Second,
the process of getting into the course changed significantly. In general, it is problematic to intentionally refrain from changing offerings.

Alternatively, we could have performed a more tightly controlled A–B study. We could have done this in a controlled lab setting, but we felt that this would not be an authentic environment and hence would lack ecological validity. We could have done this on the course level, but felt would be unethical to essentially withhold early grade information to half the students. Ultimately, we felt that a cross-year comparison provided the most study utility, without compromising our moral imperative to not hurt students.

**Threats to external validity** Edwards and Shams [22] recently characterized a corpus of student-authored test suites as being short (only one student wrote more than 21 test cases), similar (89% of students wrote exactly 21 test cases), and ineffective (their test suites missed a “significant proportion” of bugs). However, the test suites we reviewed (produced both with and without the aid of Examplar\(_a\)) were generally long (students wrote an average of 39 test cases), varied significantly in length (some students wrote more than 200 test cases), and were highly effective at catching bugs.

This contrast leads us to believe there are manifold unaccounted factors that significantly affect students’ ability to write tests. The suites we studied were produced in an environment differing from Edwards and Shams in population, course level, prior experience, language, problems, pedagogy, and tooling. A holistic understanding of these factors is essential to moving our understanding of testing pedagogy out of its infancy—but our significantly different outcomes should provide an incentive for doing so.

However, our students’ high baseline of performance does not, on its own, explain our positive results. Having some familiarity with the mechanics of

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Chaffs</th>
<th>% Final Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Examplar(_a)</td>
<td>Final</td>
</tr>
<tr>
<td>DocDiff</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>FileSystem</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Updater</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>JoinLists</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>MapReduce</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Aggregate</td>
<td>26</td>
<td>57</td>
</tr>
</tbody>
</table>
writing test cases is, undoubtedly, a prerequisite of effectively using Examplar, and our students’ high baseline of performance suggests they meet this prerequisite. (Aptitude with testing is also literally a prerequisite of csci0190: to enroll, students must complete a sequence of projects in the preceding summer that involve testing.) Yet, to benefit from Examplar’s feedback, students must:

1. write either invalid or unthorough test cases, then
2. evaluate those test cases with Examplar, and then
3. act effectively upon Examplar’s feedback.

The improvements in validity effected by Examplar indicate that our students met both of these conditions — was it because they were precocious? Probably not. It seems very unlikely that less advanced students would be less prone to problem misconceptions than advanced students. And, as we will discuss in chapter 9, Examplar still proved helpful to students lacking a correct notional machine for its feedback.

Unfortunately, we cannot entirely rule out the possibility that students elected to use Examplar because of participant bias. We defend against this possibility with several (incomplete) mitigations: (1) Examplar was not presented an experimental intervention, (2) its usage telemetry was unavailable to both course staff and students, (3) it was fairly slow and inconvenient to use. Nonetheless, the students selected for admission by Brown University are almost certainly better-than-average at appeasing their instructors. Our observations that students nearly-universally used Examplar on their own volition, then, should not be considered a major contribution of this work; rather, they should be interpreted

<table>
<thead>
<tr>
<th>Assignment</th>
<th>2017 Correct</th>
<th>n</th>
<th>2018 Correct</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocDiff</td>
<td>63.7%</td>
<td>91</td>
<td>74.7%</td>
<td>75</td>
</tr>
<tr>
<td>Nile</td>
<td>59.5%</td>
<td>74</td>
<td>68.6%</td>
<td>70</td>
</tr>
<tr>
<td>AddingMachine</td>
<td>38.2%</td>
<td>76</td>
<td>54.1%</td>
<td>61</td>
</tr>
<tr>
<td>Palindrome</td>
<td>86.8%</td>
<td>76</td>
<td>88.5%</td>
<td>61</td>
</tr>
<tr>
<td>SumLargest</td>
<td>84.2%</td>
<td>76</td>
<td>91.8%</td>
<td>61</td>
</tr>
<tr>
<td>Filesystem</td>
<td>77.6%</td>
<td>76</td>
<td>62.9%</td>
<td>62</td>
</tr>
<tr>
<td>Updater</td>
<td>28.0%</td>
<td>75</td>
<td>36.4%</td>
<td>66</td>
</tr>
<tr>
<td>JoinLists</td>
<td>64.1%</td>
<td>39</td>
<td>75.8%</td>
<td>33</td>
</tr>
<tr>
<td>MapReduce</td>
<td>69.2%</td>
<td>39</td>
<td>63.6%</td>
<td>33</td>
</tr>
<tr>
<td>Aggregate</td>
<td>63.2%</td>
<td>622</td>
<td>68.2%</td>
<td>522</td>
</tr>
</tbody>
</table>

Table 8: For each of the comparable assignments and in aggregate: the proportion of the n final implementation submissions for each year that were correct.
as a counterpoint to the notion students, broadly, “won’t test”, and as supporting
evidence for the validity of our simple cross-year comparison.

6.4 INDIRECT BENEFITS

In addition to the quantifiable improvements we observed in students’ final
submissions, Examplar\textsubscript{α} provided a host of other benefits:

REDUCED LOAD ON COURSE STAFF On the assignments for which we could
not provide Examplar\textsubscript{α}, course staffers reported an uptick in questions that they
felt could have been resolved by Examplar\textsubscript{α}; this also shows up in data gathered
about the use of hours [54] §6.1.2.

MORE ROBUST AUTOGRAADING Providing Examplar\textsubscript{α} instances forced the
course staff to finalize the wheats for each assignment before the assignment went
out to students. This process uncovered major issues in four assignments, before
they were released to students.

TEACHING UNDERSPECIFIED BEHAVIOR Underspecification was not a learn-
ing goal of the course. However, it appears that some students did gain an
understanding of what underspecified behavior is via their use of Examplar\textsubscript{α}.
We received several Piazza posts in which students discovered they were testing
underspecified behavior, e.g.:

My test below was rejected by a wheat. I think this might because
I’m checking for unspecified behavior ie. when an empty string is
passed into the content of a file. Are we meant to assume that an
empty string can never be passed into the content of a file?

6.5 OVER-INCENTIVATION?

Students seemed to enjoy Examplar\textsubscript{α}’s gamification immensely (e.g., fig. [11]).
Yet, on one assignment, FileSystem, the proportion of correct implementations
declined precipitously in 2018. We attribute this decline to differences in the time
allotted: students in 2018 were allotted half as much time as students in 2017.
Nevertheless, both the validity and thoroughness of test suites for this assignment
improved in 2018. We believe Examplar\textsubscript{α} may have monopolized students’ time
with test development—benefiting their test suites, but to the detriment of their
implementations.
Figure 11: Examplar_α received attention on a university Facebook page for anonymous admiration.

We can adjust Examplar_α’s “game” via our selection of chaffs, but this needs experimentation. Too few, and students may prematurely conclude that they are “done” with testing (and, crucially, problem comprehension). Too many, and students may divert too much time to Examplar_α and too little towards developing their implementation. Finding this balance is therefore essential future work. Students, themselves, may be poor judges of this balance. The survey instrument described in section 6.1.2 additionally asked students if they would have preferred Examplar_α to have featured fewer, more, or about the same number of chaffs; 60.9% of respondents answered “more chaffs”, 34.8% answered “about the same number of chaffs”, and only 4.3% of respondents answered “fewer chaffs”.
Figure 12: ExamplarB provides a unified editing environment for example-writing, implementation, and testing. Once the student begins their implementation, ExamplarB executes the student’s tests against their implementation. The above student has conflated median for mean in their implementation, but not in their test cases. Accordingly, ExamplarB informs them that their tests are valid, but that one of these valid test cases has rejected the student’s own implementation. The failure of a valid test implies a bug in the student’s implementation.

In controlled environments where students can be repeatedly reminded to follow metacognitive scaffolds, students have higher productivity, self-efficacy, and independence [9]. Where students are forced to solve input–output examples before beginning their implementations, they may produce better solutions [23].

1 This chapter adapts content previously published by the author in Will Students Write Tests Early Without Coercion? [1].
and make fewer errors [24]. However, left to their own devices students may lack the metacognitive awareness to realize they even need to apply these scaffolds. Students who are encouraged to follow the Design Recipe in lecture may not formulate any examples or test cases and consequently struggle [21]. Requiring the final submission of test cases on assignments will coax students to write them, but not necessarily well: A study by Edwards and Shams [22] of students trained in test-driven development and graded on test suite coverage found that students’ tests were both few and uninformative; most students wrote exactly as many tests as there were methods, and those tests tended to only evaluate the “happy path” of their respective methods.

Given the bleak research literature, it is unsurprising that recent work [23, 24] has explored (with success) forcing students to solve input–output examples before allowing them to begin their implementation work. Yet, there is also cause to not abandon hope of students’ ability to self-regulate: in chapter 6, we demonstrated that students can produce numerous, high-quality tests given the right assessment methodology, and that, when aided by helpful feedback, will even write tests when not required to do so by the assignment. With the right incentives and support, students can be coax to test—but can they be coaxed to test early?

We could not assess this question with Examplarα alone. The fact that Examplarα was a separate IDE from the one that students used for implementation meant that we had insufficient telemetry into the implementation process to explore how students interleaved example-writing and implementation effort. This separation also introduced considerable friction and complexity into Examplarα’s usage model (see section 7.1).

To rectify these issues, we developed Examplarβ (pictured in fig. 12), which provides a multi-file, unified editing environment for example-writing, implementation, and testing: each click of “Run”, regardless of which file is “open” provides feedback about the quality of the students’ tests, and their own test results on their implementation (if available).

7.1 Complexities in Examplarα’s Usage Model

CPO (and, by extension, Examplarα) provides a single-file editing environment. To develop their implementation, a student must open up a CPO browser tab to the appropriate code.arr file. And, to develop their test suite, the student must open a CPO browser tab to the appropriate tests.arr file. This arrangement lends itself to a simple execution model: in each case, clicking Run always executes the code edited in that tab, and the REPL is always situated in the context of the top-level of the edited file. It also introduces context-switching cost: one cannot
productively initiate a run in one tab, then switch to another, as several popular web browsers (including Google Chrome) severely throttle Javascript execution on background tabs; the run will seemingly never complete so long as one is not actively watching it. The introduction of Examplar$_a$ further increases the cost of context-switching: CPO and Examplar$_a$ cannot be simultaneously to edit one’s tests.arr file; only one tab of these can be open at a time, thereby subjecting context-switching students to Examplar$_a$ and CPO’s considerable loading times.

Unfortunately, context-switching between files and tools and crucial for getting meaningful feedback, as feedback is perversely dependent on editing environment. While editing one’s implementation, it is extremely valuable to have easy access to the results of running one’s tests on that implementation. Yet, to get this feedback, the student must click Run in the context of their CPO instance open to tests.arr. Then, having context-switched to that instance, one might be tempted to augment their test suite with additional tests. Yet CPO cannot provide feedback about the validity of one’s tests; only Examplar$_a$ provides that!

These dependencies between editing environment and feedback require students internalize that:

1. to develop one’s implementation, open CPO to code.arr,
2. to test one’s implementation, open CPO to tests.arr, and
3. to develop one’s tests, open Examplar$_a$ to tests.arr.

…and to possess the metacognitive sophistication to reason about which kind of feedback they need, and to possess the grit required to overcome the context-switching costs involved in accessing that feedback.

7.2 INTEGRATING TESTING AND IMPLEMENTATION

Examplar$_b$ seeks to reduce the cost of context-switching, to gently encourage students to begin with example writing, and to eliminate the dependency between editing environment and feedback.

7.2.1 Reducing the cost of context switching.

To reduce the cost of context-switching, Examplar$_b$ implements a tabbed, multi-file editor, presenting buffers for example-writing/testing (tests.arr), for implementation (code.arr), and a third buffer for any dependencies of both code and tests (common.arr). Examplar$_b$ does not provide a general-purpose multi-file editor; rather, each Examplar$_b$ instance is limited to loading these (up-to) three files associated with each assignment. These files are loaded automatically upon
opening Examplar\textsubscript{\textbeta} for an assignment, copied from instructor-authored templates in a public Google Drive folder (see fig. 13).

### 7.2.2 Encouraging students to begin with example-writing.

In the CPO-and-Examplar\textsubscript{\textalpha} workflow, the options of beginning with example-writing or implementation are presented with equal weight: the assignment handouts merely present students with two hyperlinks to starter files (tests.arr in Examplar\textsubscript{\textalpha}, and code.arr in CPO), for which we possess little control over which students will click first. With an assignment-aware, multi-file editor we have the opportunity present students with a single link—to Examplar\textsubscript{\textbeta}—and then manipulate the user interface of Examplar\textsubscript{\textbeta} to give preference towards beginning with example writing.

Examplar\textsubscript{\textbeta} hides the code.arr file until students click a \begin{center} \textbf{Begin Implementation} \end{center} button, pictured in fig. 15. The button is replaced by a code.arr tab thereafter. And, regardless of whether the student has previously clicked BEGIN IMPLEMENTATION, the tests.arr file remains \textit{always} the initially displayed file each time the IDE is opened.

### 7.2.3 Making feedback independent from environment.

As a consequence of providing a multi-file editing environment, we must resolve:
1. What does it mean to click Run?

2. What does the REPL do?

To eliminate the dependency between open file and feedback provided upon run, the answers to these questions should be uniform, independent of which file is visible at the instant Run is clicked or an expression is executed in the REPL. That is, a student working on their code.arr file should not need to remember to switch to their tests.arr file before clicking Run to receive feedback about the quality of their implementation.

Figure [14] illustrates Examplarβ’s evaluation model. Regardless of which file is actively visible in the editor, Examplarβ uses the students’ tests.arr file as the basis of all execution. Assignment template files are prepared such that the code.arr file provides all of its bindings to dependents (i.e., provide *), and that the tests.arr file includes all bindings from code.arr (i.e., include my-gdrive(“code.arr”)).

As with Examplarα, Examplarβ begins by executing the students’ test suite against each of the wheat implementations. While the wheats are sequentially injected, Examplarβ tallies which, if any, of the students’ tests have failed (and are thus invalid). If any invalid tests are present after wheat execution completes, those invalid tests are reported to the student without revealing why the tests failed (pictured in fig. [15]) and thoroughness is not subsequently assessed.

If a wheat fails because an unexpected error was encountered in a check block, the invalidity is reported (as in fig. [16]) and the further execution of wheats ceases.

Next, if all tests are valid, Examplarβ then assesses the thoroughness of the suite by running its tests against each chaff. As shown in fig. [17], chaffs are represented with bug icons, which are shaded blue when the chaff is caught; mousing over the chaffs highlights the tests that rejected it.

As with Examplarα, Examplarβ aims to discourage students from using the REPL to discover the correct output of functions. If the student has yet to click Begin Implementation, Examplarβ next executes the students’ test suite against a “dummy implementation”—a wheat in which every function body has been stubbed out with raise(‘output hidden’). Executions of the function-under-test in the REPL consequently fail with an “output hidden” error message, as shown in fig. [18]. Whereas Examplarα disabled the REPL entirely, this dummy implementation allows Examplarβ to provide a REPL prior to the availability of the student-authored implementation.

Alternatively, if the student has begun their implementation, Examplarβ skips the dummy implementation and instead assesses the student’s implementation with their tests.arr suite. Examplarβ displays these results adjacent to the validity–thoroughness feedback of tests.arr, as pictured in fig. [12]. If the student
Figure 14: Upon each click of “Run”, Examplar\(\beta\) successively runs the student’s tests against instructor-authored wheat and chaff implementations. If the student’s tests are valid, Examplar\(\beta\) additionally runs the student’s tests against their own implementation.
7.2 Integrating Testing and Implementation

Figure 15: While the wheats are sequentially executed, Examplarβ tallies which tests failed (and are thus invalid). If any invalid tests are detected, the invalid tests are reported to the student (without revealing why the tests failed), and thoroughness is not subsequently assessed.

Figure 16: If an unexpected error halts any check blocks in the tests file from executing, the execution of wheats ceases and the invalidity is reported.

Figure 17: If all tests are valid, Examplarβ then assesses the thoroughness of the suite by running its tests against each chaff. In Examplarβ’s feedback, chaffs are represented with bug icons, which are shaded blue when the chaff is caught. Mousing over the chaffs highlights the tests that rejected it.
Figure 18: Exemplar-B discourages using the REPL discover the correct output of functions. Prior to the clicking of Begin Implementation, REPL executions occur in the context of a “dummy implementation”—a wheat in which every function body has been stubbed out with `raise('output hidden')`.

has written implementation-specific tests in their `code.arr` file, these results are accompanied by a message that the validity and thoroughness of these tests are unknown.
WILL STUDENTS TEST WITHOUT COERCION?\(^1\)

In this chapter, we design a novel measure to assess students’ adherence to examples-first development. We apply these measures to students who used Examplar\(\beta\) in a semester-long accelerated introductory computer science course, and find high voluntary adherence, especially relative to the literature’s low expectations.

8.1 PEDAGOGIC CONTEXT

We assess the submissions of roughly sixty students in the Fall 2019 offering of csci0190. This section describes the salient qualities of the course; the course, its assignments and the availability of executable example feedback are described more fully in chapter 5.

DEMographics The enrollees were mostly first-year students, many of whom had prior computing experience; all had to pass a set of programming exercises to gain entry. Enrollment declined slightly over the course of the semester: 64 students submitted the first assignment; 59 submitted the final assignment.

PEDAGOGY Students were encouraged, but not required, to follow the Design Recipe—unless they sought help, at which point course staff would expect to see all the steps and help with the earliest incomplete one.

8.1.1 Assignment Structure

The course was project-oriented, featuring 17 programming projects.\(^2\) For each of these assignments, students were required to submit a code file containing their implementation, a tests file containing their test suite, and a common file. The

---

\(^1\) This chapter adapts content previously published by the author in Will Students Write Tests Early Without Coercion? [1].

common file was a shared dependency of both the tests and code file and provided a place for course staff to provide common definitions, and for students to write definitions useful to both their tests and code files (e.g., a helper function).

**Grading**  As with other offerings of csci0190, students’ final test suites were primarily graded on the basis of their validity and thoroughness.

8.2 Prior Art

Kazerouni et al. \[55\] propose a family of metrics to assess the *balance* and *ordering* of students’ testing efforts and implementation efforts of students writing Java. In these metrics, “effort” is quantified by the number of line-level changes between file-saves (as measured by *git diff*), and is reported at three levels of granularity: project, work-session, and method.

Unfortunately, line-based metrics are inherently sensitive to syntactic quirks. Minor stylistic preferences between students may be reflected as substantial differences in effort. For instance, where one student might write:

```plaintext
let foo = if a { b } else { c };
```

...another might, *equivalently*, write:

```plaintext
let foo;
if a {
    foo = b;
} else {
    foo = c;
}
```

It seems unlikely that the latter takes *six times* as much effort to write as the former.

Stylistic differences between students aside, line-based metrics may also misrepresent an individual student’s balance of work between testing and implementation. In JUnit (the testing framework used by students in Kazerouni et al.’s work), the syntactic forms associated with implementation work are very similar to those associated with tests. However, some languages (like Pyret, which we use) have concise testing syntax, making “lines” incomparable.

Buffardi & Edwards \[56\] assessed students’ adherence to test-driven development by computing each student’s test coverage (against their own implementation) averaged across their submissions to an external automated assessment system. Unfortunately, if submitting to the automated assessment system is tedious, students may leave their IDE to do so only infrequently—this interval of observation is thus potentially too infrequent to compose an adequate picture of a student’s incremental progress.
8.3 Discretizing Effort

To assess students’ testing and implementation efforts over time, we must define both an interval of observation, and a unit of effort. We derive both of these from clicks of the Run button.

The intervals created by successive runs provide a meaningful unit of effort: a programmer clicks Run with the frequency at which they wish to receive feedback. Whatever amount of testing or implementation a student completes between two successive runs reflects, by definition, the amount of testing or implementation effort which that student was comfortable undertaking on their own before requesting feedback again. In contrast, file saves only reflect the frequency at which students wish to preserve their work.

To quantify a student’s balance of implementation versus testing effort, we might simply contrast the number of test-and-run intervals to the number of implement-and-run intervals. However, if students tend to both implement and test within the same run-intervals, these intervals will be too coarse to be informative: test-and-run intervals and implement-and-run intervals will be one-and-the-same. We posit that students tend to compartmentalize their work inside run-intervals to either testing or implementation—rarely both.

8.3.1 Compartmentalization of Effort

We therefore begin by asking: do students tend to compartmentalize their work-between-runs to one of either testing or implementation? Yes. To produce this answer, we instrumented our editor to track the files modified within each run interval, and then counted and compared the number of runs occurring after each of the eight possible combinations of file modifications:

\[
\begin{align*}
\{\emptyset\} &= 7111 & \text{6.45\% of runs followed no modifications} \\
\{\text{code}\} &= 54026 & \text{83.73\% of runs followed modifying one file} \\
\{\text{tests}\} &= 34087 \\
\{\text{common}\} &= 4148 \\
\{\text{code, tests}\} &= 5480 & \text{9.82\% of runs followed modifying multiple files} \\
\{\text{code, common}\} &= 1745 \\
\{\text{tests, common}\} &= 2191 \\
\{\text{code, tests, common}\} &= 1403
\end{align*}
\]
Run-intervals in which students edited multiple files were rare: among the 98,932 run-intervals which included edits to students’ text or code files, only 6,883 (6.96%) entailed edits to both.

8.3.2 Effort Across Assignments

How did effort vary between assignments? To assess this, we should not assume that effort, as measured by sheer number of run-intervals, is directly comparable between students; different students might simply tend to click Run with different frequency.

We begin by establishing, for each student, baselines which characterizes their “usual” number of testing and implementation run-intervals: that student’s average number of implementation and testing intervals across all assignments, and the standard deviation of those quantities for each assignment. We then plot, in fig. [19] each student’s relative quantity of implementation and testing intervals for each assignment, measured in units of standard deviations from that student’s mean. (We exclude two assignments for which we failed to log data, and the three partner assignments.)

The testing and implementation effort involved in each assignment seems substantially impacted by the character of the assignment. For instance, the three assignments (SORTACLE, ORACLE, and MST) in which students implemented testing oracles (programs that test other programs) uniformly involved fewer-than-typical implementation and testing intervals. (A possible factor: these assignments do not have a binary notion of correctness, and thus integrated validity-thoroughness feedback was not available for them.) Another trio of similar assignments, TWEESearch1, TWEESearch2, and TWEESearch3, involved similar distributions of effort.

8.4 Examples-First Adherence

8.4.1 When do students click Begin Implementation?

Oftentimes immediately. Of 703 logged run sequences in which students clicked Begin Implementation,\textsuperscript{3} 363 (51.64%) clicked Begin Implementation before clicking Run for the first time. Of these, 165 (45.45%) actually edited their code file within that same initial run-interval. This suggests that about half of students

\textsuperscript{3} There were 4 logged sequences in which students never clicked Begin Implementation. Three of these sequences occurred on assignments completed with a partner. We presume that, in these cases, the partner did the implementation work.
Figure 19: Each point reflects the relative testing and implementation effort of a student, measured in standard deviations from their typical testing and implementation efforts. Implementation effort ranges on the x-axes; testing ranges on the y-axes. Assignments are parenthetically numbered with the index of their appearance in the course.
who clicked \texttt{Begin Implementation} immediately did \textit{not} necessarily do so to begin their implementation.

Without a population to compare to, we cannot say with any certainty that hiding the code file behind a \texttt{Begin Implementation} button swayed students towards initial testing. Nonetheless, we are encouraged by these data. We anticipated that most students would reflexively click \texttt{Begin Implementation} immediately, yet about half of students did not. Of those who did, about half did not actually edit their implementation within that run-interval. This suggests that many students may be clicking \texttt{Begin Implementation} just to survey the initial contents of the code file.

These data point to a possible design improvement: if students wish to be able to see the code file, make its contents initially visible but un-editable (until students click \texttt{Begin Implementation}).

8.4.2 \textit{How thoroughly do students test prior to their implementation efforts?}

A student who adheres to an examples-first programming methodology will develop \textit{interesting} examples \textit{prior} to their implementation efforts. The examples must be interesting, because uninteresting assertions do not probe one’s understanding of the problem. This example-writing must occur prior to implementation, because writing input–output assertions \textit{after} implementation is merely testing—tests \textit{confirm} implementation correctness; examples \textit{anticipate} it. Consequently, a good metric of examples-first adherence should:

- Evaluate the \textit{quality} of examples—not the quantity: the measure should not reward uninteresting assertions, and should be unaffected by the volume of edits to the tests file.

- Reward a student’s authorship of interesting examples \textit{prior} to implementation: interesting examples written after implementation contribute less to problem understanding.

- Not penalize students for using tests to \textit{review} their implementation efforts: modifications to tests after implementation efforts should not contribute negatively.

These properties are satisfied by the \textit{mean implementation-interval thoroughness} (\texttt{miit}, for short): the mean of the peak thoroughness achieved prior to each implementation interval. Concretely, consider this (synthetic) sequence of run-intervals:

\[
\left[ \frac{1}{5}, \frac{2}{5}, \frac{2}{5}, \frac{2}{5}, \frac{2}{5}, \frac{2}{5}, \frac{3}{5}, \frac{3}{5}, \frac{3}{5}, \frac{3}{5}, \frac{3}{5}, \frac{3}{5}, \frac{5}{5} \right]
\]
Figure 20: The MIT of each student (represented by points) on each assignment. The shaded areas with fractional labels reflect each possible level of thoroughness as the proportion of buggy implementations rejected. Underlaid box-and-whiskers plots summarize the overall MIT on each assignment.
The fractions denote the thoroughness feedback resulting from that run, and ⌗ denotes that the run resulted in an error; □ denotes modifications to tests, and △ denotes modifications to code. Now, consider only the implementation-intervals:

\[
\left[ \frac{2}{5}, \frac{2}{5}, \frac{3}{5} \right]
\]

This student completed 3 impl-intervals after achieving a peak thoroughness of $\frac{2}{5}$, and one more after achieving a peak thoroughness of $\frac{3}{5}$. Their miit is therefore $\left( \frac{2}{5} \times 3 + \frac{3}{5} \times 1 \right) / 4 = 0.45$.

We use the peak thoroughness achieved prior to each implementation interval (as opposed to the last thoroughness achieved) because where thoroughness declines, it is usually because the student has written a very large test suite and has commented most or all of it out to either focus on a particular test result, or to temporarily hasten their edit-and-run cycle.

Figure 20 visualizes students’ miit for each of the non-partner assignments with validity–thoroughness feedback. (We exclude the partner assignments, because we cannot always combine the logs of cooperating students into a single consistent timeline of work.) We hoped that providing students with integrated feedback on their examples would guide them to achieve some level of thoroughness prior to their implementation efforts. However, mindful of the frustration that ensues when students are unable to catch all the buggy implementations, the instructor told them to try to catch “most of them” but move on once they had done so instead of getting bogged down. Indeed, students typically voluntarily achieved a moderate level of thoroughness before the bulk of their implementation work: the miit of the median student ranged from $\frac{2}{5}$ to $\frac{5}{6}$. Only a handful of students on each assignment (six, on average) did not achieve any thoroughness before the bulk of their implementation work.

8.5 LIMITATIONS

The primary contribution of this chapter is a novel metric, miit, for assessing how “test first” students are; its secondary contribution is the application of these metrics to analyze approximately 700 editing sequences produced by students. Neither of these contributions are without limitations.

LIMITATIONS OF METRIC This metric is sensitive to chaff selection. This metric is subject to a ceiling effect if the chaffs are too easy to catch. Conversely, it is subject to a floor effect if the chaffs are too difficult to catch. How this metric should be interpreted is highly dependent on how chaffs are selected. For instance, if the chaffs comprise of subtly buggy implementations, a student’s
MIT will reflect how carefully they tested for subtle bugs before the bulk of their implementation work. As we do not consider this to be an important aspect of problem understanding, it was important that our chaffs did not include subtly buggy implementations.

Limitations of application We did not observe in our students many of the bleak results logged by prior work: we found that most students did not begin their implementation work immediately, and that the vast majority wrote moderately thorough examples before their implementation efforts. However, absent further study, we cannot causally link our IDE modifications, specifically, to our positive results. Consider, for instance:

- Examples-first development is less rigid than test-driven development (which prescribes a strict interleaving of testing and implementation) and thus perhaps easier to adhere to.
- Pyret has a very lightweight, native syntax for test cases.
- The functions implemented by students are “pure”, and thus more easily testable than if they involved side-effects.

Still, our work serves as a preliminary (positive) assessment of students’ self-regulation abilities, as an example of how researchers might consider whether IDE disaffordances have warped student behavior, and as a toolbox for future analyses of students’ example-writing behavior.
WHEN IS EXAMPLAR INSUFFICIENT?¹

As we discuss in chapter⁴ researchers have adopted a variety of strategies for the purpose of getting students to understand problems before starting to write solutions. Several of these amount to having students solve *examples of expected behavior* (e.g., in the form of unit tests) before starting to code. Research also shows [23, 24] that these sorts of methodological guard-rails are quite successful.

However, unlike these approaches, Examplarα and Examplarβ do not enforce a particular work-flow onto students, nor did we impose a particular work-flow in our classroom deployments of these tools: usage of Examplarα was explicitly optional and Examplarβ’s executable example feedback could be ignored or silenced through a variety of means, including a “skip tests” setting! Rather, we present Examplar to our students as a kind of teaching assistant (TA). This TA has a very limited interface: it can only answer questions about the input–output behavior described by the problem specification, and it only expresses its answer in terms of wheat-passing and chaff-catching. However, it is always present, responds immediately, and is infallible.² So, when students have questions about the problem specification, we urge them to “ask Examplar first”. What impact, then, does Examplar have on student help-seeking?

Specifically, given the presence of this infallible oracle of (un)specified behavior, we are interested in learning:

- **What questions do students still have left about (un)specified behavior?**

- **What is the influence of automatic, on-demand feedback about (un)specified behavior in these questions?**

To answer these questions, we conducted an exploratory evaluation of these issues in the context of a post-secondary CS course. We manually reviewed the 1,247 assignment-related student posts in the online help-forum of this course,¹

---

¹ This chapter adapts content previously published by the author in *Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors* [3].

² This claim needs clarification. We mean this in two ways. First, human TA staff sometimes provide incorrect or contradictory information, whereas all students run the same Examplar. Second, we stipulate that the behavior of the wheats is definitional, i.e., so long as student work is consistent with the wheats, we will accept that behavior as correct.
filtering down to a set of 298 questions relating to the specified input–output behavior of assignments. We iteratively reviewed these postings, noting whether they related to specified or unspecified behavior, related to the input–output of functions, mentioned IDE feedback, and other noteworthy themes. We identify a number of intriguing pathologies: students asked questions that could have been answered with automated feedback, misinterpreted feedback, developed faulty mental models of the feedback mechanism, and possessed faulty preconceptions about assignment specifications. These pathologies suggest faults in both our pedagogy and the design of Examplar.

9.1 RELATED WORK

Given the importance of problem understanding, this chapter examines situations where Examplar-style automation is not enough. Our closest related work is other projects that have examined student help-seeking. While student help-seeking is a longstanding area of inquiry in education research \[57\] research into computing students’ help-seeking on course forums is comparatively nascent.

Marwan et al. \[58\] assess the unproductive help-seeking behaviors of students working in a programming environment that provides on-demand hints. Similarly, prior research on Examplar has assessed students’ usage patterns of the tool. In contrast, our work evaluates help-seeking outside of the programming environment, and the subject matter of the help students sought.

Vellukunnel et al. \[59\] evaluated forum posts from 395 students enrolled in CS2 courses across two universities. The researchers manually labeled work-related posts by quality and content, and found that some categories positively correlated with grades. However, they do not offer detailed insight into the nature of the content-clarification posts, which is our sole focus here.

We focus on forum posts but ignore help sought in TA hours. We chose to not monitor these hours because students might find that invasive and become reluctant to seek help. Ren et al. \[54\] studied TA use through a non-intrusive but lightweight mechanism. The trade-off is that their instrument cannot provide more information into our question, and they anyway find that relatively few questions about input–output behavior are asked during hours.

We are also laser-focused on a particular kind of help-seeking: questions relating to the input–output behavior of the problem. We are not aware of much literature on this particular topic. Ren et al. \[54\] observe that a cohort of students trained in the Design Recipe \[8\] frequently ask questions about the earliest step in the Design Recipe (which is focused on problem reinterpretation), and suggest that this “shows that TAs helped students with understanding the problem, not only focusing on the end products.” However, the second and third steps of the
Design Recipe also focus on problem reinterpretation, and TAs were seldom asked about these steps. Much of the literature on programming problem solving finds that students may get entrenched in problem misunderstandings without realizing it [4,6,5], and thus, presumably, cannot seek help for those misunderstandings.

Examplar bears some resemblance to intelligent tutoring systems [60]: unlike a “normal” development environment, Examplar has domain knowledge of the assignment being solved by the student and can alert students to their misconceptions. In this work, we introduce Examplar to students as an alternative to soliciting help from course staff: a student who formulates a question about a problem’s input–output behavior can, in principle, pose that question as a test case to Examplar—rather than to course staff—and receive an immediate answer. However, Examplar is not a cognitive tutoring system [61]; it is purely reactive to students’ inputs, and does not adapt to students’ learning progressions.

9.2 PEDAGOGIC CONTEXT

The pedagogic context of this research is the Fall 2020 offering of csci0190. This section describes the salient qualities of the course; the course, its assignments and the availability of executable example feedback are described more fully in chapter 5.

DEMOGRAPHICS Its enrollment was 114 students, of whom 92 were first-years and about 20% female.

ASSIGNMENTS The Fall 2020 offering of csci0190 had 15 programming projects (and no exams).

In this chapter, we focus on 14 of the 15 assignments, because one does not fit the scope of our analysis: the final assignment, named “24”. In this multi-stage assignment, students were not given an input–output specification. Rather, they are presented with a series of computational word problems to solve. Henceforth, when we mention the course’s assignments, we will be referring only to the 14 for which students produced an implementation from an assignment specification.

GRADING As with other offerings of csci0190, students’ final test suites were primarily graded on the basis of their validity and thoroughness.

STUDENT HELP RESOURCES The Fall 2020 offering had 15 undergraduate TAS who cumulatively provided 30 hours per week of one-on-one live assistance to students, and answered questions posted to an online help forum (as did the
professor). Questions posed to the help forum were, as a rule, only visible to course staff; students could not see the questions posed by other students. When, occasionally, course staff changed the visibility of a student’s post to ‘public’, the identity of that student remained hidden to other students.

Testing feedback—both via Examplar (against a small set of chaffs), and from the testing auto-grader (against a much larger suite of chaffs)—was provided on most but not all assignments.

**Impact of COVID-19** The pandemic had several impacts:

- The course was entirely virtual. (Usually, it is conducted in person, with most students resident on campus.)

- The enrollment was nearly twice normal (due to an unusual configuration of courses in the department) but with only a 25% increase in course staff.

- The start of the school year for first-year students was moved to Spring 2021, but incoming students were permitted to take one course (for free) in the Fall. Thus, for most students, this was their only course.

- Normally, students meet with TAs in person and value the interaction (often waiting in long lines to do so). However, TA sessions were also made virtual (and staggered to handle time zones).

Due to some combination of the above, students posted many more forum questions than before: 1,602 posts, which (after accounting for class size difference) is 54% more than in Fall 2019, for largely the same assignments.

### 9.3 Navigating (Un)Specified Behavior

Automated grading demands that students accurately match the behavior described by each assignment’s specification. To succeed, students needed to precisely interpret both the specified and unspecified behaviors of each assignment. As many of our findings (section 9.5) relate to how students responded to (un)specified behaviors, we briefly review the challenges they pose in this section.

**Unspecified Behavior** The unspecified behaviors are those aspects for which the programmer has discretion over the exact behavior of their program. For example:

1. A specification of arithmetic mean might only define the behavior over non-empty lists of numbers, but a particular implementation will do something (defined exception? o? -1? division-by-zero?) for empty inputs.
2. A list can have more than one most-frequent element. The specification of the mode function may not fully indicate which one(s) should be produced.

A black-box test of unspecified behavior is a kind of invalid test. If an implementation (say an instructor-provided one) fails against them, that doesn’t make the implementation wrong! Thus, it is important to identify such tests as invalid. Examplar does this via multiple wheats (recall section 2.1.3).

**Resolving invalidity** However, what a student should do after being told a test is invalid is complicated. It may be that the test reflects a misunderstanding of the specified behavior of the problem, in which case the student should correct the test. Or, it may be that the test provides an unspecified input (e.g., the mean of an empty list), in which case the student should remove the test.

Or, most subtly, it may be that the test involves *semi-specified* behavior, in which case the student should use *property-based testing* (pbt) [62]. For instance, a thorough test suite of the mode should include inputs with multiple modes, but it should not assert that the output for such inputs is any one value—that would be an implementation-specific test. Rather, it must check that the output satisfies a property: that it is a mode of the input list.

**Pedagogy** To prepare students to handle semi-specified behavior, we devote two assignments, Sortacle and Oracle, entirely to pbt. For these assignments, students’ sole objective was to produce a testing oracle, a function that consumes a function purportedly implementing a specification, and produces true if it is a correct implementation, and false if it is buggy.

We reference the first of these non-traditional assignments, Sortacle, throughout our findings. In Sortacle, students implemented a predicate that consumes two lists of Persons (records with a name and an age) and produces true if the second list is an instance of the first sorted by age; otherwise false. Here, the precise ordering of Persons with the same age is intentionally left unspecified.

**9.4 Methods**

We began by automatically collecting students’ postings to the assignment-related categories of the course’s online forum. This collection comprised 1,247 of the approximately 1,602 postings made across all categories. These assignment-related posts discussed the behavior of students’ programs, the behavior described by the assignment, the runtime-complexity of functions, programming language issues, course procedure, and more. We retained only the posts relating to input–output behavior of assignments for further examination; i.e., posts that concerned a
function specified by the assignment handout, and where the source of confusion could be observed with a test relating that function’s input to its output. This second criterion included both postings in which the question was phrased theoretically (“What should X output when given Y?”), and postings where the question was about the students’ code without mention of the specification (“Why is this test invalid?”). In all, 298 postings related to the input–output behavior of an assignment.

Next, we reviewed these input–output-related postings on both a per-assignment and cross-assignment basis for common (or otherwise notable) phenomena. On all but Sortacle and Oracle, we additionally attempted to classify whether each posting was prompted by Examplar, and whether it related to unspecified inputs, semi-specified outputs, or the specified behavior of the assignment. We considered a posting to be prompted by Examplar if the student mentioned Examplar explicitly, included a screenshot of its feedback, or alluded to its feedback (e.g., “Why is this test invalid?”). A posting related to unspecified inputs if it concerned an input outside the domain of the assignment; it related to semi-specified outputs if it concerned an input admitting multiple possible outputs; and it related to the specified behavior of the problem if it concerned a case in which the input is specified and admits one valid output. We left particularly hard-to-classify postings—nine, in all—unclassified.

This classification work was mostly performed by one author; inter-rater reliability methods are therefore inapplicable. For hard-to-classify postings, this author sought feedback from the former TAs of the course. The other author of this work—who was the instructor of the observed course—provided context and interpretation for the patterns identified by the first author. As this is interpretation innately subjective, we center the presentation of our observations on the quoted postings of students.

The severity of input–output-related postings varied widely. At one extreme, course staff merely nudged the student to re-read a portion of the assignment handout. At the other extreme, the staff needed to make changes to the automated feedback mechanisms to correct bugs. Regardless of severity, they reflect some of the self-insurmountable problems encountered by students: ones where the student either could (or did) not answer their question from the assignment specification or Examplar.

We pay special attention to the ninth assignment, TweeSearch. It asks students to implement a search function that consumes a list of “tweets” and a query to fuzzily search for, and produces a list of tweets sorted by relevance. The exact output order of equally-relevant tweets is unspecified, so there can be several valid answers. Examplar will reject as invalid any test that asserts an exact order. Students were told to treat this like an exam: “without consulting course staff
except for critical issues (broken links, possible assignment typo, etc.). Therefore, any input–output behavior questions about this assignment would have been posted as a last resort.

9.5 Observations

We now try to answer the research questions posed in the abstract and introduction. We begin with basic statistics about the kinds of questions asked, then move on to several observations that we believe may be insightful to researchers and educators. The 298 postings we reviewed were authored by 90 different students. The most prolific of these students authored 12 input–output related posts; the median student authored three.

9.5.1 Distribution of Questions

We begin by examining the distribution of the 230 input–output related questions students posed across 12 programming assignments (we exclude the 68 postings relating to SORTACLE and ORACLE) across several basic axes.

Specified or Unspecified? There is a big difference between (what a student believes to be) specified versus unspecified behavior. If the behavior appears to be specified, a student is more likely to ask why the outcome is or is not what it is. If it seems unspecified, the student does not know what should happen at all. We found 106 to be about specified behavior and 110 about unspecified; a few could not be definitively classified, or included questions of both kinds.

Kinds of Unspecified Behavior On most assignments, students grappled with one of two kinds of unspecified behavior: inputs for which no output was specified, and inputs for which the output was only semi-specified. Among the 110 postings that included questions about unspecified behavior, 58 concerned unspecified inputs, and 60 concerned semi-specified outputs.

Relationship to Automated Feedback Within the ten assignments for which students had automated feedback about the validity of their tests, students posed 169 input–output-related questions; of these 105 mentioned automated feedback that a test was invalid. Of these 105 questions, 28 concerned specified behavior, 29 concerned unspecified inputs, and 43 concerned semi-specified outputs.
9.5.2 Not Using Examplar’s Feedback

Despite the availability of Examplar, students posed questions that could have been answered automatically with a test case; e.g.:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do the movement functions in \texttt{Updater} need to be repeatable/stackable?</td>
</tr>
</tbody>
</table>

For questions like the above, we might attribute help-seeking on the course forum instead of Examplar to a preference for prose. One student, asked why they had not answered their question using Examplar, clarified that they had been pen-and-papering out examples and had not yet realized validity feedback was available.

Other cases are more puzzling. In several instances, students asked the TAs whether a particular test case was valid or not:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should we be allowed to update multiple nodes or navigate around the tree without turning the cursor back into a tree between updates. For example, would a test like this be consistent with the problem specification?</td>
</tr>
<tr>
<td>[test case code elided]</td>
</tr>
</tbody>
</table>

This question could have been answered by running this test case in Examplar, but the student does not mention having tried it. Is it possible that they did not know Examplar’s validity feedback corresponds to “consistent with the problem specification”? Or, could it be that students tended to prefer interacting with course staff over automated tooling? For further examples of this pathology, see the student postings reprinted in section 9.5.6.

9.5.3 Input Bias

When faced with feedback that a test was invalid, students needed to determine whether the invalidity was caused by an unspecified input, or an over-constrained assertion on semi-specified output. Multiple students mis-attributed the invalidity of their over-constrained tests to unspecified inputs.

Recall that in \textsc{TweeSearch}, testing for a particular output order of equally-relevant search results is invalid. However, at least six students misattributed the cause; e.g.:
I just wanted to clarify: can we assume that two tweets in our list will never have the same overlap with a search tweet? I think the answer is yes since Examplar errors no matter the order of the outputted list, but I was unsure because it doesn’t error if you check the length of the outputted list. Regardless, I just wanted to clarify since I don’t think it’s specified in the assignment page.

This student misattributed their test invalidity to the input, and thus asks if they can (incorrectly) assume that inputs that admit equally-relevant results are invalid.

Such misattributions are worrying. They might push students to delete their over-constrained tests, rather than to reformulate them to be property-based. Much worse, students may end up implementing an incorrect solution that is brittle in the face of valid inputs that they have incorrectly ruled out.

This posting is notable in that the student correctly intuits the method for distinguishing between invalid inputs and semi-specified outputs: use PBT. Other students were not so fortunate, and even this student only presents this as a passing observation, not recognizing that this is in fact what they are expected to do. This, at least, suggests a pedagogic flaw in the class.

9.5.4 Failure to Transfer

Even though students had completely two assignments devoted to PBT, on postings for later assignments with semi-specified outputs, course staff frequently had to prompt students to remember how they had handled this situation in the past. This happened both soon after those assignments, and late in the semester.

On TweeSearch, which came two weeks after the latter PBT assignment, students needed to adapt the property-based testing for sortedness that they developed in Sortacle. Unfortunately, TweeSearch stumped a significant proportion of students: 23 of the 36 input–output-related posts on this assignment concerned testing sortedness. Several students drew a partial connection to PBT, but struggled to find a middle-ground between exact-value testing and PBT; for instance:
This student has clearly read the specification closely, identified that exact-value testing is inappropriate, and succeeded with weak pbt—but they were unable to commit to this strategy. (The post in section 9.5.3 also shows this.)

These failures are worrisome, and suggest difficulties that students face with testing against properties rather than concrete outputs. Unfortunately, there is little research on this topic; the only paper we know of [63] reports fairly positive outcomes, and does not contain the kind of fine-grained observations we are making. This clearly identifies the need for more detailed future work.

9.5.5 Not Understanding Examplar

Some students’ postings indicated that they did not have a clear execution model for the validity feedback produced by Examplar, even late into the semester. This confusion was often evident in their response to such feedback (e.g., individually testing every possible output of a semi-specified problem), but was sometimes explicit. We saw three main categories of confusion about Examplar’s model.

what is checked? As an example, on ContinuedFractions, the eighth programming project of the course, one student asked:

So when I was creating tests in examplar, I tried testing threshold as followed below but received “These tests do not match the behavior described by the assignment” [...] I don’t know if it is my implementation or something to do with an invalid test. It would be great if someone cleared me on where I seem to be going wrong.

Feedback that tests are invalid can only indicate an issue with tests. Because the student’s tests are run against instructor implementations, and the tests can be run before the student’s implementation has even begun, invalidity cannot
possibly be a statement about the student’s implementation. We unfortunately see similar misunderstandings late into the semester too.

**MIS-EXPERIMENTS** Of the 23 postings to TweeSearch about its semi-specified output behavior, 16 posts explained that they had individually tested every possible ordering—all of which were (correctly) rejected as invalid by Examplar. Reactions included bug reports; e.g.:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>I know I’m supposed to treat TweeSearch like an exam, but I think there’s something broken. The following checks all fail, which is interesting because they seem to cover every single possible output. Am I missing something, or is the given implementation broken?</td>
</tr>
</tbody>
</table>

Another student mixed PBT (good!) with exact-value tests (which would never work) of every possible combination of outputs:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>I cannot figure out the order in the case that two tweets have the same similarity scores, and I assumed you wouldn’t tell me, so I just tried testing it out. Is the highlighted part not a logical paradox here?</td>
</tr>
</tbody>
</table>

The comment about the “logical paradox” reflects a very interesting misconception about the nature of testing and problem definition.

**REIFYING THE MODEL** Recall that Examplar\(\alpha\) reported the number of wheats it uses to check validity, and Examplar\(\beta\) “simplified” its interface by removing this information. Sadly, several postings suggest that hiding this detail of the execution model may have been counterproductive; e.g.:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>If our tests are only being tested against one wheat, then something weird is happening. If there are deliberately multiple wheats to ensure that our tests are order-agnostic, then it’d be good to know. Does that mean we have to write our own predicate “equality” function that ignores orders for tweets with the same score?</td>
</tr>
</tbody>
</table>

This student had formed a perfect conception of the situation and, had they been given the wheat count, they would have come to exactly the right understanding without needing to ask.
9.5.6 Specification Preconceptions

Postings regarding the specified behavior of functions often suggested the student carried preconceptions about the intended behavior of the assignment.

**Imagined functionality** Students wondered whether they needed to provide functionality not mentioned by the assignment specification. For instance, the second assignment asked students to implement a recommendation engine by analyzing frequently-paired books. Two students wondered about the case-sensitivity of the engine; e.g.:

*asked...*

When matching titles, do we assume they match exactly, or should we take into account capitalization? Should we also take other words into account? For example, if I recommend "The Lightning Thief by Rick Riordan", does the function need to be able to tell that it's the same thing as "The Lightning Thief", "lightning thief", "the lightning thief", "Lightning Thief", etc?

The assignment specification does not suggest that the recommender should do anything other than check exact matches, but such functionality would be useful in a real-world version of the assignment. Similarly, on the first assignment, which implements a very simple text similarity checker:

*asked...*

Should the program consider punctuation when evaluating overlap?

**Imagined constraints** On Sortacle, in addition to building the checker, students were required to read and reflect on the article “Falsehoods Programmers Believe About Names” [64]. Nonetheless, among the 43 input–output-related questions posed by students, more than a dozen concerned the well-formedness of names and ages; e.g.:

*asked...*

Does a name have to start with a capitalized letter? Does a name have to contain a first and a last name? Does it matter if the randomly generated string doesn’t make actual sense in terms of English? (e.g. Do “Shbs”, “Xusnhy” count as names...?)

*asked...*

Is it appropriate for our names to have symbols in them?
All of these were effectively answered by the handout (which says they are all valid names, and hence invalid constraints).

This tendency extended, to a lesser extent, to more abstract assignments. On Updater, two groups of students wondered about validity constraints on trees; e.g., one wrote:

**asked...**

For a single node, is it guaranteed that its children will have distinct values?

There was nothing in the specification suggesting that the data carried by sibling nodes ought to be unique.

**Mathematical Language**  
In an assignment where students implemented a palindrome-checker on strings, three students wondered if the empty string ought to be considered palindromic. As one student explained:

**asked...**

Would an empty string be a palindrome? I’m tempted to say yes, for the same sort of reason that an empty set is a subset of every set; but there’s also something about that that offends against the ordinary English conception of a palindrome.

9.5.7  *Expanding the Specification*

Whether through prior conditioning or other reasons, many students seem to not entirely understand, or perhaps even believe, that the assignment constitutes
a kind of contract, and that the course staff will not demand, and often do not even care about, aspects not specified. For instance, on a problem to compute text overlap, the function was only specified on non-empty lists (since the formula does not make sense otherwise). Nonetheless, students wanted to know the “right” thing to do for empty inputs, and asked several questions along the lines of:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should we account for if the inputted lists are empty in overlap? If so, would overlap be 0 or a message?</td>
</tr>
</tbody>
</table>

Theoretically, unspecified inputs ought to be a reduction of work for students: their use helps keep specifications shorter, and they reduce the surface area of functionality that students must implement and test. In practice, however, these postings suggest that unspecified inputs may be a source of anxiety, at least early in the semester.

9.5.8 Tooling Pain Points

We identified three pain-points in the tooling around Examplar:

**error suppression** Upon a click of Run, Examplar first executes the students’ tests against a set of correct implementations. Only if these tests are valid does it then run them against the students’ implementation. If there are any invalid tests, those tests are highlighted, but information about why they are invalid is suppressed to discourage students from merely copy-pasting the “correct” output result into their test suite.

In at least one instance, this measure hid valuable information from the student: they had, incorrectly, written `g.names` instead of `g.names()` in a test case. The resulting error message was suppressed, and the student hypothesized that an unrelated issue was to blame.

**undetectable invalidity** On JoinLists, students implement a function for sorting an input list according to a given comparator function. The specification dictated that this comparator must define a total order, but it is generally impossible to detect whether a purported comparator satisfies this property. As such, students were able to write invalid tests that were marked as valid.
9.6 Limitations

No lightweight relational testing  JoinLists also illustrates a problem we noticed in multiple places. In that assignment, the reduce function’s output is only semi-specified. One student asked:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>So I wonder, is there a way for us to do something like:</td>
</tr>
<tr>
<td>j-reduce(lam(x,y): x - y end,</td>
</tr>
<tr>
<td>[join-list:1,2,3]) is 2 or -4</td>
</tr>
<tr>
<td>(btw, we know the above example doesn’t work... since we cannot use “or” in check box...we think)</td>
</tr>
</tbody>
</table>

This student correctly observes that the output is a set. They are hoping the language would let them write multiple outputs (or -4) just as easily as they can write one (2). The language does not support that directly, and it requires a non-trivial change to the test case for the student to specify the set of acceptable outputs instead.

9.6 Limitations

Threats to internal validity  Our view of student help-seeking was limited to interactions in the online course forum; we do not know what questions students asked course staff in office hours. Our classifications of postings (e.g., of whether questions concerned unspecified behavior) should be treated with caution: they were mostly performed by one author, and some hard-to-classify postings remained unclassified. Our interpretations of postings are best-guesses, informed by our experience with the course. Although these guesses will inform our future interventions and research, they are, at this time, untested hypotheses.

Threats to external validity  In this work, we report on a number of challenges that our students encountered which we had not anticipated. Different student populations, especially those working under different conditions (e.g., differences in assignments, instruction, help availability, grading incentives, pandemics, etc.), would likely encounter different challenges, and other instructors would probably vary in the pathologies that they failed to anticipate or found notable. We therefore caution readers against drawing broad conclusions about novice programmers from our observations.

This work only examines problems that, while sophisticated, for the most part: do not involve state or other side-effects; have agreed-upon data structures; and have a clear goal. Removing each of these makes “automated teaching” much harder or even impossible. Stateful interfaces will almost certainly make the writing of
unit tests far harder. If the assignment itself requires the design of data structures (like our Updater does), then the data structure must be abstracted into interfaces that can be tested against. Finally, entirely open-ended problems (e.g., asking students to create a game of their choice) are not amenable to this method at all, which assumes there is some objective specification that can be checked.

9.7 Discussion

To some extent, this chapter validates the use of tools like Examplar. Examplar’s intent is to serve as a guardrail for students, alerting them to their problem misconceptions before final grading. Many of these instances we cannot see from our current dataset, but the 105 postings that cited Examplar’s feedback reflect a lower-bound for the number of instances where a student was stopped from veering off-track. This suggests the value of “automated tests” like Examplar.

However, questions that Examplar could answer (i.e., questions about input–output behavior) were a minority of the questions posed by students about their assignments. There is thus considerable opportunity in expanding the range of feedback offered by pedagogic IDEs to also address questions about style (e.g., as Learn-OCaml [65] does) and questions about implementation [66].

Postings also suggest that many students did not know how to respond to feedback about invalidity. We observe a time-consuming anti-pattern too, reminiscent of Prather et al.’s [5] observations of students consumed by automated feedback on their implementations: students reacting to invalidity by exhaustively testing every possible output of semi-specified functions. These students were not as adequately prepared for testing unspecified behavior by Sortacle and Oracle (the course’s testing-focused assignments) as we had hoped. And, at least one student who did understand Examplar’s assessment model for unspecified behavior was hampered by the tool’s opaque presentation of feedback (“If there are deliberately multiple wheats [...] it’d be good to know.”). Future research must address how to help students understand respond productively to invalidity, especially that caused by unspecified behavior.

More broadly, problem comprehension seems to demand new skills that are not very well covered in current curricula or standards. For instance, early examples and subsequent testing both require good adversarial thinking: a mindset that is not usually included in definitions of “computational” thinking but is clearly relevant even from an early stage (not only in areas like security). Yet, the barriers to employing adversarial thinking may be more social than instructional or technical: the availability of Examplar did not stop students from posing questions about the expected input–output behavior of problems (including questions that used prose to precisely describe illustrative test cases). Future
work should consider the possibility that students prefer the assurances of human course staff over that of automated feedback.
WHEN ELSE DOES EXAMPLAR WORK POORLY?

While Examplar proved highly beneficial in the primary context studied (various offerings of csci0190, in chapters 6, 8 and 9), it is not a cure-all. As discussed in section 2.2 Examplar’s assessment model substantially limits the scope of assignments for which wheat–chaff feedback can be supplied. In this chapter, we consider how the pedagogic context of Examplar may also substantially affect its utility.

CSCI 0111 Computing Foundations: Data\(^1\) (csci0111) is the first course in a relaxed, three-course introductory sequence. The Spring 2020 offering of the course featured ten programming projects; the first six in Pyret, and the remaining four in Python. The second, third, and fourth assignments used Examplar\(_β\).

In Spring 2020, we conducted a think-aloud, A/B study among csci0111 students, in which participants were assigned a programming problem to solve in 20 minutes using the Examplar\(_β\) IDE. Among seven participants, four received wheat–chaff feedback; three did not. The programming problem was followed by additional time for questions and discussion about Examplar. The primary goal of this research was to study the differences between students who had wheat–chaff feedback available to them, and those that did not. Instead, the study (particularly the discussion following the thinkaloud) revealed a variety of environmental factors that confounded the study and severely undermined Examplar’s usefulness for all participants:

10.1 WHEN THE RUN IS CLICKED INFREQUENTLY.

Examplar provides wheat–chaff feedback upon each click of Run. The usefulness of the environment hinges on how early and often Run is clicked; if students seldom click Run, or first click Run late into their development process, they will seldom get feedback, or only get feedback late into their development process.

This threat to Examplar’s utility was on striking display during in the csci0111 think-aloud. All participants’ first click of “Run” (and thus their first exposure to wheat–chaff feedback, if available) occurred very late into the development

---

\(^1\) https://cs.brown.edu/courses/csci0111/spring2020/
process: P1 at 15:30, P2 at 19:41, P3 at 7:00, P4 at 14:22, P5 at 15:08, P6 at 15:38, and P7 at 15:30. P3’s comparatively early run at seven minutes occurred prior to the modification of any files, and was used to clear the REPL; their next click of run occurred at 20:27. P7’s first run at 15:30 was only to clear the REPL; their next run came at 21:34. As participants were not told in advanced whether wheat–chaff feedback would be available, it was not until students’ first non-erroring Run that they learned whether or not wheat–chaff feedback was available.

This late use of Run was exacerbated by a development style favored informal testing in the REPL to use of check blocks. P1, P2, and P7 strongly preferred the REPL to check-block-based testing. P2 used the REPL to confirm their understanding of the support code. P1 and P7 mostly used it to confirm their understanding of their own helper functions.

10.2 WHEN FEEDBACK IS MISALIGNED WITH INCENTIVES.

When Examplar’s feedback is misaligned with the grading incentives of the course, students might feel more frustrated by it than helped.

We anticipated that our csci0111 participants, like the students we studied in csci0190, would positively regard Examplar’s feedback (e.g., see section 6.5). Indeed, we observed several utterances suggesting that participants were motivated by chaffs to write more tests:

1. P3: “I have to catch that third buggy.”

2. P5: “[If I had chaffs], I’d be running this test against the correct code, and if I catch the buggies, I know i’m doing the right thing”

3. P6: “Oh wait! I’m only catching one of three buggy!?”

However, two participants described substantial chaff-related frustrations. P7 described themselves as motivated to test by chaffs, but also deeply frustrated by them (and testing in general). On the matter of chaffs, they complained about the opacity of feedback. As a student taking the class for a grade, they felt like they (and their grade-taking friends) had to catch all chaffs (or they would fail) — while acknowledging that a TA probably told them it was alright to miss a few chaffs. They complained at length on the frustration of thinking of nonsensical (with respect to the real-world context of the problem) edge conditions, but acknowledged that the chaffs had pushed them to learn to think about educations far better than they had on the first few assignments.

They also expressed great frustration at hitting testing quantity targets. Their final words while working on rainfall were:
Oh I caught all three [chaffs]? I would still probably write a bunch of baloney tests because I feel like they like seeing I tested a bunch, but i would be doing that because I wanted to get a good grade in this class and not for any other reason.

Similarly, remarking on validity and thoroughness feedback, P1 said:

[...] they made a very big deal about how to test how to test rigorously in the beginning and then it was just general understanding were supposed to test all this stuff. But there wasn’t really feedback on tests. And I don’t think there’s a feedback on the homework really, you just get your grade. And so there’s like points off and I’m not sure why there are points off. I guess I could go ask.

Indeed, csci0111, unlike csci0190, did not primarily grade students on the validity and thoroughness of their test suites. In csci0190, students’ final submissions are primarily automatically graded on the correctness of their code, and the validity and thoroughness of their test suites; Examplar’s feedback thus provides a glimpse into final grading: The same wheats are used between Examplar and final grading, and the chaffs used by Examplar are a strict subset of those used in final grading. Absent this alignment between feedback and grading methodology, Examplar’s value to students is fairly abstract.

10.3 WHEN THE IDE IS UNFAMILIAR.

The A/B study was confounded by unfamiliarity with the IDE, particularly the multi-file paradigm. Examplar requires that black-box tests are authored in a separate file from the problem implementation. In Examplarβ, this separation is realized as distinct editor tabs. All participants in the csci0111 think-aloud received at least two reminders of this separation: once written in the problem handout, and once in my verbal summary of the handout. Three participants nonetheless stumbled, reflexively beginning their implementation in the tests tab:

1. P1 opened Examplar at 2:30, I gave “two tabs available” reminder, at 3:00 they began writing the ‘rainfall’ skeleton in the tests tab, then switch to handout to try to find ‘official’ signature, at 3:51 I reminded them that a function template was available in the code tab and they finally click ‘Begin Implementation’.

2. At 1:54, P2 began writing the ‘rainfall’ skeleton in the tests tab, I tell them that template is already present in the tests tab until 19:41, at which point they hit run for the first time, and get a very gnarly shadowing error—I
explain they are working in the wrong tab. At 21:43, they paste their impl into the code file without deleting the template.

3. At 2:25, P4 began writing the ‘rainfall’ skeleton in the tests tab. At 2:49 I interject and point out a skeleton is already available in the code tab, and they exclaim, “Oh! What do you know! I should read all of this before I start.” However, they then (3:07) turn back to the handout and start scanning for a ‘rainfall’ template. I clarify explicitly about clicking “Begin Implementation”, and only then do they click it.

Additionally, in discussion, P3 remarked:

This structure of having these two files was never explained to me, so
I like don’t know how it works. I don’t know why they’re in different tabs.

A pre-participant remarked that they were not used to having template code already present for them—in contrast to csci0190, the non-Examplar assignments of csci0111 do not provide a CPO instance with starter code. A few participants asked if they needed to copy in the helper function code snippets from the handout. (They did not.)

10.4 WHEN TESTS ARE DIFFICULT TO WRITE.

Although the perceived value of Examplar’s feedback may incentivize students to write input–output examples, it does not entirely negate factors that disincentivize this kind of example-writing. For instance, on csci0190’s FluidImages assignment, formulating new input–output examples requires students to painstakingly write out image pixel data; perhaps consequently, this assignment garnered the second-fewest clicks of “Run Tests” of any assignment during the 2018 offering of csci0190 (see fig. 10).

A student may also struggle with writing input–output examples if they have little experience doing so. In csci0111, students typically implemented programs to analyze existing, given tables of data, rather than author their input data. For two participants in the csci0111 think-aloud, this was a barrier to example-writing and programming progress:

1. At 3:00, P7 remarked “my first gut reaction is how do I actually look at that data I have, because I see that it’s this,” and then moused over the g-drive import of the support code. I clarified this problem was like Homework 6, where they wrote functions working over lists, not tables.

2. A 6:53, P3 moused over the g-drive import of the support code and remarked “let’s remember how to access data from this thing” — perhaps confusing
the support code import with an import of a Google Sheet. A 8:44, when writing their plan, they included the step “figure out how to get the data lol”. At 18:28, they moused over the g-drive import again and remarked “How do you get the data?”, then exclaimed “Oh, do I just have to do import as some shit?” At this point, I interjected that the necessary imports were all in place, to which they responded “But how do I access this list? [...] In my head, I should have, like, imported-table = blahblahblah”.

In both cases, an expectation that input data was be provided prevented the student from easily formulating input–output examples or tests.
CONCLUSION

Providing students with timely feedback of their input–output examples incentivizes them to develop input–output examples, improves the quality of their test cases, and may improve the quality of their implementations. Realizing these benefits, however, requires both the selection of an appropriate model for assessing the quality of examples, and the implementation of the model in a manner that effectively communicates assessments to students. The key contributions of this dissertation are:

1. the observation that example-writing cannot deliver its purported benefits without additional feedback,
2. a mechanism that provides this feedback by assessing example quality,
3. proof-of-concept implementations of this mechanism in an IDE,
4. experimental evidence that this feedback was both compelling and helpful to students,
5. a catalogue of pathologies and design considerations that future instructors and researchers must account for in any applications or extensions of my work.

The classifier model of test suite assessment (chapter 2), combined with a concept-oriented approach to chaff development (appendix A) is one approach with which instructors can provide example feedback. We demonstrated (chapter 6) the viability of this model with Examplarα (detailed in chapter 3), a development environment standalone from students’ usual implementation and testing environment, which is specialized for writing examples. This IDE cannot be used for developing or testing one’s implementation (students needed to complete these tasks in the usual Pyret development environment), and was (after the first assignment) completely optional for students to use. Nonetheless, virtually all students used Examplarα extensively on every assignment. The validity of students’ test suites was drastically better than prior years, and the correctness of students’ implementations improved slightly (though not significantly).
These findings assured us that Examplar$_\alpha$ was, at the very least, non-harmful. Subsequently, we developed a successor IDE, Examplar$_\beta$, which integrated Examplar’s feedback into an environment in which students could also develop and test their implementations (chapter 7). This integration, in principle, reduced the degree of self-regulation required from students to benefit from Examplar’s feedback. It also gave us further visibility into how students balanced their example-writing, implementation, and testing efforts. We developed a measure of examples-first-ness (chapter 8) that instructors using our assessment model can apply to evaluate how thoroughly their students explore problem specifications with examples before the bulk of their implementation work. Our preliminary assessment of editing patterns of csci0190 students with this metric indicated that these students, on their own volition, moderately explored the specification via examples before the bulk of their implementation work.

11.1 CAVEATS AND CONSEQUENCES

However, Examplar is not a panacea. In chapter 2, we noted that Examplar’s assessment model renders it useless for certain classes of problems, including those where “correct” and “buggy” is not a strict binary. In chapter 6 we observed that having a robust test suite does not necessarily translate to a correct implementation: on an assignment where the allotted days was halved between 2017 and 2018, the quality of students’ test suites was seemingly unaffected by the abbreviated timeframe, but the quality of their implementations declined precipitously. In chapter 8, we observed that, while Examplar$_\beta$ effectively forewarned students of invalidity, students were sometimes unsure of how to interpret that feedback, and needed to seek additional help from the course staff. And, in chapter 10, we noted a slew of other pathologies relating to the habits of students and the circumstances of Examplar’s deployment; e.g., that Examplar cannot provide feedback until students click Run.

These caveats are not merely theoretical considerations; they are critically important qualities of both our assessment model and of Examplar that instructors adopting either must weigh, and they are opportunities for vital future work.

11.1.1 Opportunities In the Assessment Model

For one, the limitations of our assessment model meaningfully constrain constrain the kinds of assignments it can be used on. In csci0190, these limitations precluded Examplar’s availability on two assignments early in the course. The consequences Examplar’s absence were not contained these assignments; this
brief gap in Examplar’s availability eroded students’ trust that Examplar would be available on future assignments (section 9.5.2). The limitations of our assessment model thus poses both practical pedagogic risks and a research opportunity: How might we help students hone their problem understanding on such assignments?

11.1.2 Opportunities in the Implementation

There is significant room for experimentation in the implementation of our assessment model, too. Our analysis of students’ course forum posts (chapter 9) uncovered numerous situations in which Examplar’s feedback prompted students to pose follow-up questions to course staff, each of which is a research opportunity: Could different automated feedback have answered this question?

COMMUNICATING (IN)VALIDITY We observed several postings suggesting uncertainty about how to interpret Examplar’s feedback about invalidity (section 9.5.5), e.g.:

<table>
<thead>
<tr>
<th>asked...</th>
</tr>
</thead>
<tbody>
<tr>
<td>So when I was creating tests in examplar, I tried testing threshold as followed below but received “These tests do not match the behavior described by the assignment” […] I don’t know if it is my implementation or something to do with an invalid test. It would be great if someone cleared me on where I seem to be going wrong.</td>
</tr>
</tbody>
</table>

Cosmetic changes to Examplar’s feedback might have averted this question; Examplar could have said, directly:

There an issue with your test suite. This test case is invalid, meaning it might reject even a correct implementation.

However, merely understanding that invalidity feedback indicates a problem in one’s test suite does not mean that one knows how to respond to that feedback. How a student should respond to invalidity feedback is complex; there are three kinds of invalidity, each requiring a distinct response:

1. the invalid test reflects a misunderstanding of the specified behavior of the problem (e.g., median([1, 1, 4]) is 2), in which case the student should correct the test, or

2. the invalid test fails to account for the specification allowing multiple, valid outputs for certain inputs (e.g., mode([1, 2]) is 1), in which case the student should apply property-based testing techniques, or
3. the invalid test supplies an input for which the output is undefined (e.g., `median([])` is 0), in which case the student should delete the test.

We encountered numerous postings suggesting that students struggled with these subtleties (section 9.5.5). Some, for instance, seemed to possess a conceptual model of invalidity that did not admit for all three possibilities; e.g.:

```
I know I’m supposed to treat TweeSearch like an exam, but I think there’s something broken. The following checks all fail, which is interesting because they seem to cover every single possible output. Am I missing something, or is the given implementation broken?
```

Another student, in a similar position, declared Examplar’s feedback to be a “logical paradox”. What alternative or additional feedback might have helped students navigate the subtleties of invalidity?

At the very least, we can imagine cosmetic changes to the IDE’s output; e.g. reporting invalidity with the message:

```
There an issue with your test suite. This test case is invalid, meaning it might reject even a correct implementation. To fix this, start by figuring out why this test case is invalid. Click here to learn more.
```

Better, yet: the IDE, itself, could automatically apply heuristics to deduce why a test case is invalid. If, for instance, the wheats communicated to Examplar that a function was invoked with an out-of-domain input, Examplar could directly inform the student that their test case had invalidity of the third kind. And if, for instance, Examplar also checked whether a student’s test case passed some-but-not-all of the wheats, Examplar could inform the student that their test case exhibited invalidity of the second kind.

**Thoroughness without fixation or frustration**  At best, thoroughness feedback provides students with a sense of how confident they should be that they understand the assignment, and motivates them to develop their confidence through example-writing (e.g., as we observed in section 10.2). However, at worst, thoroughness feedback can spark both fixation and frustration. The precipitous decline of implementation quality — but not test suite quality — on Filesystem submissions in 2018 (section 6.5) might be explained by Examplar inducing students to devote too much attention to catching chaffs, and too little attention to ironing out their implementation. And, since Examplar does not provide any hints as to how a chaff is buggy, students can only (and often do, in csci0190) seek help from course staff when faced with a stubborn, uncaught
chaff. Worryingly, some students frustrated by uncaught chaffs might not seek help at all, and suffer privately.

Since Examplar, itself, does not implement any safeguards that prevent these pathological cases, we rely on course staffs’ careful application of chaff curation best practices (appendix A). The fragility of this approach compels us to question: *How could the implementation of thoroughness feedback be improved?* For performance reasons, we recommend that Examplar is deployed with no more than ten chaffs (appendix A.5), but it may be that this small number encourages too much fixation on particular chaffs — would an IDE that selects chaffs dynamically, or can quickly execute dozens of chaffs, induce less fixation or frustration from uncaught chaffs? Or, the IDE could, for instance, provide hints about why a chaff is buggy. Or, the IDE could allow students to ‘mute’ stubborn, uncaught chaffs from the IDE’s thoroughness feedback. This design space is enormous.

### 11.1.3 Opportunities in Configuration

Several of our recommendations for chaff curation (appendix A) are rooted in intuition, not data. For instance, our recommendation that Examplar is deployed with no more than ten chaffs (appendix A.5) stems from a concern that any more would, cumulatively, be “too slow to run” or “too fatiguing to catch”. *Could these performance and difficulty constraints be estimated more rigorously?*

We also recommend that instructors applying our assessment model select chaffs whose bugs reflect conceptual issues, rather than careless programming errors (appendix A.2). This recommendation is rooted in an assumption that artisanal, conceptually-flawed chaffs are more effective at revealing student misconceptions than other approaches of chaff curation. However, curating such chaffs demands significant effort from instructors, which may discourage them from adopting example feedback wholesale. *How effective, comparatively, are simpler, programming-error-based chaffs at assessing student understanding?*

This recommendation also stems from (1) our postulate that early feedback about one’s problem understanding is more important than early feedback about how well one’s test suite catches careless programming errors, and (2) our assumption that the set of chaffs used in Examplar must be fairly small and thus instructors should prioritize conceptually-flawed chaffs over chaffs with careless programming errors. However, the feedback provided by both kinds of chaffs is clearly useful: students are probably as liable to make careless programming mistakes as they are to have problem misconceptions! *How might an Examplar-like environment improve their test suite’s detection of programming errors?*
Lastly: There are significant opportunities in exploring the circumstances in which Examplar-like tools are deployed. For one, our large-scale studies of Examplar were limited to a single course context, csci0190, and the efficacy of Examplar in that context was undoubtedly affected by the characteristics of csci0190. For instance, the assessment model used to grade students’ tests in csci0190 was closely aligned with the assessment model employed by Examplar. In contrast, the assessment model used for grading by csci0111 was not aligned with Examplar’s assessment model, and csci0111 students voiced confusion and frustration about this misalignment (section 10.2). The impact of these kinds of synergies and syzygies warrants further investigation. *How important is it to align IDE-provided feedback with the grading mechanisms of the course?*

The circumstances of Examplar’s deployment in csci0190 also limited our research opportunities in that context. Our initial study of Examplar (chapter 6) was the only one in which we were able to reasonably compare between Examplar’s non-users (the 2017 edition of csci0190) and users (the 2018 edition of csci0190). The results of that study dissuaded us from withholding Examplar’s feedback from students in later editions of the course, and the accumulation of small assignment revisions prevented us from drawing fair comparisons between future editions of csci0190 and its 2017 edition. Consequently, in chapter 8 we could not compare the example-firstness scores of students exposed to Examplar’s feedback to the scores a cohort without Examplar’s feedback. This key question thus remains open: *To what degree does exposure to validity and thoroughness feedback influence students’ editing behavior?*
TENETS OF CHAFF CURATION

More than an editing buffer, Examplar is a conversational partner: When a student submits their tests to Examplar, the IDE replies with validity and thoroughness feedback. It can inform the student whether their tests are valid or invalid; if their tests are valid, it can additionally inform them whether they have caught or missed each of the assignment’s chaffs. However, while Examplar might be a wittier interlocutress than any other text editor the student has used, the automated TA lacks some of the basic competencies of its fleshy colleagues. Examplar has no heuristic for frustration, nor any capacity for compassion. It does not possess discretion, much less exercise it. Examplar (in its infinite hubris) does not seek help from other course staff when its students need it.

Given these limitations, the role played by course staff in the configuration of Examplar is critical: their curation of chaffs moulds the conversations students will have with Examplar. With each chaff, the instructors prime Examplar to remind students “there is something you are not yet testing”. Whether students regard these reminders as insightful or frustrating hinges largely on the nature of the chaffs. Unfortunately, we can offer few hard-and-fast rules for chaff selection. In stark contrast to wheats, the curation of chaffs is almost entirely a creative exercise. Yet, what began, for us, as a small set of intuitions (see section 2.1.3), has since been honed by hard lessons in the years following Examplar’s initial deployment. This appendix reifies those intuitions and lessons into eight key tenets for chaff curation. There surely are more lessons yet-to-be-learned.

A.1 FAVOR COVERING THE API

Strive to ensure that for each item (e.g., function, constant, etc.) defined by the assignment, there is a chaff that implements that item in a flawed manner. For instance, an assignment that defines five ‘public’ functions should have at least five distinct chaffs (see Avoid chaffs with multiple bugs), each implementing a buggy version of one of the five functions. A missing chaff — provided it is not to
difficult to catch (see \textit{Avoid difficult-to-catch chaffs}) — then signals to the student that they’ve yet to explore some well-defined facet of the specification.

This tenet must be balanced with \textit{Avoid long-running wheats \& chaffs}, since an assignment should not have so many chaffs that Examplar is unable to provide timely feedback. If full coverage of the API is impossible, prioritize covering the items where logical errors are likeliest to occur.

\subsection*{A.2 \textsc{Favor logical errors over programming errors}}

Each chaff represents a single opportunity to communicate to the student that a facet of the assignment is not yet represented in their test suite. Unfortunately, because you\textsuperscript{1} must \textit{Avoid long-running wheats \& chaffs}, you are typically limited to at most ten chaffs in Examplar. This limited number of communication opportunities is best expended on guiding students towards identifying problem misconceptions, rather than helping the student develop a test that is sensitive to subtle programming errors. While it is important that students eventually develop test suites that are highly capable at detecting programming errors, ensuring problem understanding has priority. For this reason, we favor flaws in chaffs that correspond to misconceptions, rather than programming errors.

This tenet must, occasionally, be balanced with \textit{Favor covering the API}, since very simple items defined by the assignment may not lend themselves to interesting logical errors. In these circumstances, we tended to err on the side of covering the API and use a simple programming error as our chaff’s flaw.

\subsection*{A.3 \textsc{Avoid chaffs with multiple bugs}}

Examplar’s vocabulary is highly limited: the “chaff missed” feedback is the sole mechanism by which Examplar informs the student that they have a gap in their tests, and it is only upon the first detection of a flaw in a chaff that feedback for that chaff changes (it transitions from \textit{missed} to \textit{caught}). Examplar cannot communicate whether additional, latent flaws remain. If finding these additional flaws is pedagogically important (and it always should be), then it is best to give those flaws their own distinct chaffs. For this reason, we strongly recommend against crafting chaffs that contain multiple flaws.

Do not compromise this tenet in order to \textit{Favor covering the API} and \textit{Avoid long-running wheats \& chaffs}. You cannot satisfy \textit{Favor covering the API} by compromising this tenet, because a single chaff cannot communicate that a student is missing tests of multiple items.

\textsuperscript{1} This appendix is addressed, in particular, to course staff seeking to deploy Examplar.
A.4 AVOID DIFFICULT-TO-CATCH CHAFFS

Again, Examplar’s vocabulary is highly limited: it cannot provide students with hints, nor seek help from course staff when students need it. In the absence of these mechanisms, students may expend excessive amounts of time trying to catch a missing chaff, then be driven to tears by frustration if they are unable to do so. The onus is on course staff to configure Examplar to minimize excessive frustration. Do not select difficult-to-catch chaffs.

Consideration must also be paid to the cumulative difficulty of chaffs. While it may be feasible to both have many chaffs and avoid long-running wheats & chaffs, the cumulative difficulty of catching all chaffs should not be overly taxing.

This tenet must be balanced with Favor logical errors over programming errors: Using a chaff with a subtle error may be justifiable, so long as that error highlights a critical conceptual facet of the problem. A chaff that implements a programming error (that isn’t pedagogically critical) should never be difficult to catch.

A.5 AVOID LONG-RUNNING WHEATS & CHAFFS

Examplar is intrusive: it interjects itself upon each click of Run to execute students’ tests against each wheat and chaff. If this interruption takes excessively long, Examplar will perversely incentivize students to either write fewer tests, or to seek feedback less frequently. Instructors should strive to ensure that Examplar’s feedback is produced swiftly for most students.

To manage total execution time, one must consider (1) how many tests students will write, (2) the total number of wheats and chaffs, and (3) the combined performance characteristics of the wheats and chaffs.

Although you wield little control over how many tests students will write, accounting for this factor is critical to estimating your performance budget. Feedback latency should be tolerable even for a very thorough test suite. The test suite used for final grading of implementations (provided that it is also capable of catching every final grading chaff) tends to be a good benchmark for performance. Be mindful of the interplay between learning and test writing. For instance, a lesson on property based testing will likely be followed by some students applying that technique to their own test suites, resulting in potentially hundreds of generated test cases.

With a benchmark in hand, you can begin selecting wheats and chaffs. The number of wheats should be aggressively minimized: it is often sufficient to have at most two wheats (see section 2.1.3). As a rule of thumb, there should be strictly fewer than ten chaffs. Of course, this must be balanced with Favor covering the API.
Additionally, avoid selecting chaffs whose flaws are computationally expensive (in the extreme: *Never use chaffs prone to non-termination*).

Examplar supports distributing wheats and chaffs both in source form, and as compiled binaries. Although source-distributed wheats and chaffs are convenient for assignment development, the wheats and chaffs should be compiled prior to deploying the assignment to students. The decision of distribution mode is sometimes, incorrectly, framed as being about whether the risk of students peering at the source code of wheats and chaffs is acceptable. Even if this risk is acceptable, wheats and chaffs should be distributed in compiled form; doing so drastically improves performance.

This tenet must be balanced with *Favor covering the API* and *Favor logical errors over programming errors*. For all but the simplest assignments, it is impossible to satisfy all three of these tenets. Carefully weigh improvements in performance against the pedagogic cost of sacrifices in coverage and inclusion of insightful chaffs.

### A.6 Avoid Non-Deterministic Chaffs

The dialogue between student and Examplar should be and be perceived as) both predictable and reliable. In the absence of non-deterministic tests, Examplar’s feedback should be strictly deterministic; an unchanged test suite should yield identical feedback each time it is run. A chaff, once caught, should remain caught. And, in keeping with *Avoid difficult-to-catch chaffs*, there should be no element of luck in catching a chaff. To achieve this, avoid non-deterministic chaffs.

### A.7 Never Use Chaffs Prone to Non-Termination

As a corollary to *Avoid long-running wheats & chaffs*, chaffs used in Examplar must *never* fail to terminate. Non-terminating chaffs create a perverse incentive for students to delete effective tests. Unless the student stops execution (which deprives them of any feedback), the student’s internet browser will, eventually, forcibly kill the tab Examplar is open in.

Assessing whether a chaff is prone to non-termination requires thinking adversarially about the interplay between the chaff’s flaw and the sort of tests students might write; see section 2.2.4 for an example where an unforeseen interaction between chaffs and tests led to non-termination.
Never use chaffs whose flaw is a type error. A chaff must conform to the signature of the specification, and only contain dynamic flaws. While one should favor logical errors over programming errors, ill-typedness is the most basic of problem misconceptions. Even a fairly rudimentary test suite will, implicitly, test the types of items by virtue of calling functions and comparing their outputs to expected values. If a student has a misconception about the type of a function and reflects that misconception in their test suite, those ill-typed tests will first reject the wheats and the chaffs will not even run. It is therefore a wasted opportunity to use the chaffs to highlight type misconceptions.


